

Volume 16

Number 2

ACTA CYBERNETICA

Editor-in-Chief: J. Csirik (Hungary)

Managing Editor: Z. Fülöp (Hungary)

Assistant to the Managing Editor: B. Tóth (Hungary)

Editors: L. Aceto (Denmark), M. Arató (Hungary), S. L. Bloom (USA), H. L. Bodlaender (The Netherlands), W. Brauer (Germany), L. Budach (Germany), H. Bunke (Switzerland), B. Courcelle (France), J. Demetrovics (Hungary), B. Dömölki (Hungary), J. Engelfriet (The Netherlands), Z. Ésik (Hungary), F. Gécseg (Hungary), J. Gruska (Slovakia), B. Imreh (Hungary), H. Jürgensen (Canada), A. Kelemenová (Czech Republic), L. Lovász (Hungary), G. Păun (Romania), A. Prékopa (Hungary), A. Salomaa (Finland), L. Varga (Hungary), H. Vogler (Germany), G. Wöginger (Austria)

Szeged, 2003

ACTA CYBERNETICA

Information for authors. Acta Cybernetica publishes only original papers in the field of Computer Science. Contributions are accepted for review with the understanding that the same work has not been published elsewhere.

Manuscripts must be in English and should be sent in triplicate to any of the Editors. On the first page, the *title* of the paper, the *name(s)* and *affiliation(s)*, together with the *mailing* and *electronic address(es)* of the author(s) must appear. An *abstract* summarizing the results of the paper is also required. References should be listed in alphabetical order at the end of the paper in the form which can be seen in any article already published in the journal. Manuscripts are expected to be made with a great care. If typewritten, they should be typed double-spaced on one side of each sheet. Authors are encouraged to use any available dialect of T_EX.

After acceptance, the authors will be asked to send the manuscript's source T_EX file, if any, on a diskette to the Managing Editor. Having the T_EX file of the paper can speed up the process of the publication considerably. Authors of accepted contributions may be asked to send the original drawings or computer outputs of figures appearing in the paper. In order to make a photographic reproduction possible, drawings of such figures should be on separate sheets, in India ink, and carefully lettered.

There are no page charges. Fifty reprints are supplied for each article published.

Publication information. Acta Cybernetica (ISSN 0324-721X) is published by the Department of Informatics of the University of Szeged, Szeged, Hungary. Each volume consists of four issues, two issues are published in a calendar year. For 2003 Numbers 1-2 of Volume 16 are scheduled. Subscription prices are available upon request from the publisher. Issues are sent normally by surface mail except to overseas countries where air delivery is ensured. Claims for missing issues are accepted within six months of our publication date. Please address all requests for subscription information to: Department of Informatics, University of Szeged, H-6701 Szeged, P.O.Box 652, Hungary. Tel.: (36)-(62)-546-396, Fax:(36)-(62)-546-397.

URL access. All these information and the contents of the last some issues are available in the Acta Cybernetica home page at <http://www.inf.u-szeged.hu/kutatas/actacybernetica/>.

EDITORIAL BOARD

Editor-in-Chief: **J. Csirik**
University of Szeged
Department of Computer Science
Szeged, Árpád tér 2.
H-6720 Hungary

Managing Editor: **Z. Fülöp**
University of Szeged
Department of Computer Science
Szeged, Árpád tér 2.
H-6720 Hungary

Assistant to the Managing Editor:

B. Tóth
University of Szeged
Department of Computer Science
Szeged, Árpád tér 2.
H-6720 Hungary

Editors:

L. Aceto
Distributed Systems and Semantics Unit
Department of Computer Science
Aalborg University
Fr. Bajersvej 7E
9220 Aalborg East, Denmark

F. Gécseg
University of Szeged
Department of Computer Science
Szeged, Aradi vértanúk tere 1.
H-6720 Hungary

M. Arató
University of Debrecen
Department of Mathematics
Debrecen, P.O. Box 12
H-4010 Hungary

J. Gruska
Institute of Informatics/Mathematics
Slovak Academy of Science
Dúbravská 9, Bratislava 84235
Slovakia

S. L. Bloom
Stevens Institute of Technology
Department of Pure and Applied
Mathematics
Castle Point, Hoboken
New Jersey 07030, USA

B. Imreh
University of Szeged
Department of Foundations of
Computer Science
Szeged, Aradi vértanúk tere 1.
H-6720 Hungary

H. L. Bodlaender
Department of Computer Science
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

H. Jürgen
The University of Western Ontario
Department of Computer Science
Middlesex College, London, Ontario
Canada N6A 5B7

W. Brauer
Institut für Informatik
Technische Universität München
D-80290 München
Germany

A. Kelemenová
Institute of Mathematics and
Computer Science
Silesian University at Opava
761 01 Opava, Czech Republic

L. Budach

University of Postdam
Department of Computer Science
Am Neuen Palais 10
14415 Postdam, Germany

H. Bunke

Universität Bern
Institut für Informatik und
angewandte Mathematik
Länggass strasse 51.
CH-3012 Bern, Switzerland

B. Courcelle

Université Bordeaux-1
LaBRI, 351 Cours de la Libération
33405 TALENCE Cedex
France

J. Demetrovics

MTA SZTAKI
Budapest, Lágymányosi u. 11.
H-1111 Hungary

B. Dömölki

IQSOFT
Budapest, Teleki Blanka u. 15-17.
H-1142 Hungary

J. Engelfriet

Leiden University
LIACS
P.O. Box 9512, 2300 RA Leiden
The Netherlands

Z. Ésik

University of Szeged
Department of Foundations of
Computer Science
Szeged, Aradi vértanúk tere 1.
H-6720 Hungary

L. Lovász

Eötvös Loránd University
Department of Computer Science
Budapest, Kecskeméti u. 10-12.
H-1053 Hungary

G. Păun

Institute of Mathematics
Romanian Academy
P.O.Box 1-764, RO-70700
Bucuresti, Romania

A. Prékopa

Eötvös Loránd University
Department of Operations Research
Budapest, Kecskeméti u. 10-12.
H-1053 Hungary

A. Salomaa

University of Turku
Department of Mathematics
SF-20500 Turku 50, Finland

L. Varga

Eötvös Loránd University
Department of General Computer Science
Budapest, Pázmány Péter sétány 1/c.
H-1117 Hungary

H. Vogler

Dresden University of Technology
Department of Computer Science
Foundations of Programming
D-01062 Dresden, Germany

G. Wöginger

Department of Mathematics
University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands

Preface

The 3rd Conference for PhD Students in Computer Science (CSCS) was organized by the Department of Computer Science of the University of Szeged (SZTE) and held in Szeged, Hungary from July 1 to 4, 2002. The members of the Scientific Committee were the following representants of the Hungarian doctoral schools in computer science: Mátyás Arató (DE), András Benczúr (ELTE), Miklós Bartha (SZTE), Tibor Csendes (SZTE), János Csirik (SZTE), János Demetrovics (SZTAKI), Sarolta Dibuz (Ericsson), József Dombi (SZTE), Zoltán Ésik (SZTE), Ferenc Friedler (VE), Zoltán Fülöp (SZTE), Ferenc Gécseg (chair, SZTE), Balázs Imreh (SZTE), János Kormos (DE), László Kozma (ELTE), Attila Kuba (SZTE), Eörs Máté (SZTE), Gyula Pap (DE), András Recski (BMGE), Endre Selényi (BMGE), Katalin Tarnay (NOKIA), György Turán (SZTE), and László Varga (ELTE). The members of the Organizing Committee were Tibor Csendes (chair), Lajos Schrettnner, Mariann Sebő, Péter Gábor Szabó, Boglárka Tóth, and Tamás Vinkó.

There were more than 100 participants and 88 talks in several fields of computer science and its applications. Beyond the Hungarian PhD schools in computer science, mainly the universities of Almeria, Spain and of Turku, Finland were represented. The talks were going in two parallel sections in artificial intelligence, automata and formal languages, computer networks, database theory, discrete mathematics, fuzzy decision support systems, information systems, optimization, picture processing, and software engineering. The talks of the students were completed by 4 plenary talks of leading scientists.

Three scientific journals, viz. Periodica Polytechnica (Budapest), Publicationes Mathematicae (Debrecen) and Acta Cybernetica (Szeged) offered students to publish the paper version of their presentations after a selection and review process. Altogether 35 papers were submitted for publication. The present special issue of Acta Cybernetica contains 10 such papers.

The full program of the conference, the collection of the abstracts and further information can be found at <http://www.inf.u-szeged.hu/~cscs>.

On the basis of our positive experiences, the conference will be organized in the future, too, hopefully with more foreign participants. According to the present plans, the next meeting will be held in July 2004 in Szeged.

Tibor Csendes and Zoltán Fülöp

Incorporating Linkage Learning into the GeLog Framework*

Tim Fühner[†] and Gabriella Kókai[†]

Abstract

This article introduces modifications that have been applied to *GeLog*, a genetic logic programming framework, in order to improve its performance. The main emphasis of this work is the structure processing of genetic algorithms. As studies have shown, the linkage of genes plays an important role in the performance of genetic algorithms. Thus, different approaches that take linkage learning into account have been reviewed and the most promising has been implemented and tested with *GeLog*. It is demonstrated that the modified program solves problems that proved hard for the original system.

1 Introduction

The *GeLog* program combines two approaches, inductive logic programming (ILP) and artificial evolution [1]. This work aims at improving the *GeLog* framework by incorporating methods that help the evolutionary algorithm to maintain a rugged search behavior without losing the ability to quickly find (local) optima. Both requirements are most relevant to noisy search spaces, which are often characteristic in inductive logic programming. This article introduces the modifications that were applied to the *GeLog* framework and presents the results of two experiments, which demonstrate that the program has been drastically improved.

The following section briefly introduces the *GeLog* framework. Section 3 explains the term linkage and introduces related approaches. The modifications that have been applied to *GeLog* are depicted in Section 4. In Section 5 some test results are presented. Finally, Section 6 concludes this article and provides a short outlook on future investigations and improvements.

2 Brief Introduction into GeLog

The *GeLog* framework is a genetic logic programming framework, an inductive logic programming system combined with an evolutionary search algorithm [1]. In-

*This work is supported by the grants of Bayerischer Habilitationsförderpreis 1999.

[†]Department of Computer Science II, University of Erlangen-Nuremberg, Martensstr. 3, 91058 Erlangen, Germany, e-mail: fuehner@iis-b.fhg.de, kokai@informatik.uni-erlangen.de

ductive logic programming is a machine learning approach, in which correlations of objects are ascertained by induction. Hypotheses are searched for and evaluated by comparing their classification results with a sufficiently large number of instances for which it is known whether their objects are correlated or not [2]. It is thus assumed that hypotheses classifying these training instances correctly will also approximate the target function well over any other set of instances. The learned hypotheses can be interpreted as PROLOG programs, since they consist of set of rules, that is, first order Horn clauses.

GeLog's data representation resembles more to that of genetic programming: the individual solutions consist of PROLOG program parts, which encode the hypotheses' rules. Thus, an individual comprises the target predicate as its left hand side and a number of disjunctions (right hand sides), all of which are conjunctions of literals. The following example demonstrates how individuals are represented in the the original *GeLog* implementation:

```
daughter(X0, X1) :- female(X0), parents(X1, X0, X1).
```

```
parents(X0, X1, X1).
```

```
female(X1), female(X0), parents(X1, X1, X0).
```

The depicted individual consists of three disjunctions (right hand sides); each disjunction contains a number of conjuncted literals and is terminated by a dot.

The pay-off of one hypothesis results from the number of correctly classified instances. Different selection operators have been implemented: Roulette Wheel Selection, Rank Selection, and Elitism (for further explanation of these operators see [3] and [4]).

Due to the non-standard data representation special recombination and mutation operators had to be implemented:

- Two recombination operators; (1) two individuals exchange entire disjunctions by single- or multi-point crossover, (2) two individuals exchange predicates by performing single- or multi-point crossover at disjunction level.
- Mutation operators; (1) insertion and deletion of literals, (2) insertion and deletion of entire disjunctions, (3) insertion of new variables, and (4) substitution of variables.

3 Linkage Learning and Related Work

The first complete theory of the dynamics and processing units of genetic algorithm was developed by Holland [5]. In his schema theorem he suggested that genetic algorithms process the search space implicitly parallel. A specific individual is also

a representative of a class of individuals that have certain gene values (alleles) in common. For example, individual 100101 represents the class of individuals with a leading '1' (denoted as 1*****); but it also represents individuals that contain two '0' alleles on second and third position (*00***), etc. Thus, by selecting individual solutions the (schema) classes which are represented by the individual gain influence. For example, if *00*** exhibits a relative high fitness, i.e., individuals that contain the specified '0' alleles are on average fitter than others are, this schema is represented more often than other schemata. A higher fitness is achieved if those parts of solutions are recombined that caused the former individuals to exhibit a higher fitness than other individuals. In other words, by combining fit schemata even fitter schemata are generated.

Based on the insights attained by the schema theory Goldberg formulated what he called the building block hypothesis [3]. He concluded that the central processing units of genetic algorithms are "short, low-order, and highly fit schemata". These entities he called building blocks. Goldberg also found that some problems are hard to solve for genetic algorithms because of difficulties in processing building blocks. Consider four building blocks: $H_1 = 1*****$, $H_2 = *****1$, $H_3 = 0*****$, and $H_4 = *****0$. Let the fitness of H_1 and H_2 be remarkably greater than the fitness of H_3 and H_4 , also let the fitness of a recombination of H_1 and H_2 (1*****1) be smaller than 0*****0 (the combination of H_3 and H_4). As the two recombined schemata exhibit a relatively high order, chances are high that they are disrupted quickly, resembling schemata $H_1 - H_4$. Since the selection probabilities for schemata H_3 and H_4 are low it is difficult for the genetic algorithm to recombine them both yielding the highly fit schema 0*****0 again.

The situation changes if the defining genes of the schemata are linked more tightly, since the probability of disruption decreases drastically. On the one hand that increases the chances of preserving the fit recombined schema, on the other hand it ensures that the unfit schema is discarded and not split into the two fit sub-schemata which lead to the deception. This is obviously a simplification of the dynamics of genetic algorithms and has been criticized for that reason (cf. [6, 7]). However, it could be shown that for many problems improving the linkage situation of building blocks also improved the performance of the genetic algorithm. It is therefore worthwhile to develop techniques that lead to tighter linkage of building blocks.

It was long assumed that individuals in genetic algorithms would eventually evolve towards tighter linkage. However, early efforts that used inversion operators to achieve tight linkage proved that selection is too powerful and thus counteracting linkage learning [8].

3.1 Messy GA

One of the early approaches that took this observation into account was the so-called *messy genetic algorithm* [9]. In addition to a "messy coding" which allowed for a reordering of the chromosome, linkage learning and selection were separated

into two phases such that selection is prevented from vitiating linkage learning. The two phases are repeated alternately increasing the order of building blocks that are processed. In the first phase all possible building blocks of the current order are generated. This explicit enumeration is very expensive ($O(2^k \ell^k)$, where ℓ is the chromosome length and k is the highest order of building blocks, i.e., the number of genes, that define a building block). After this enumeration the threshold operator tries to select individuals such that only those compete that define the same class of schemata. For example, 00* and 11*, but not 0*1 and *01. The second phase resembles to a simple genetic algorithm. A variation of this approach replaces the expensive enumeration of all building blocks of a specific order by a probabilistic technique [10]. Instead of generating all order- k schemata explicitly, this technique makes use of the fact that one bit string may contain multiple schemata at the same time, since the bit string is normally longer than the order of the schema. Thus, only a fraction of the former $O(2^k \ell^k)$ individuals had to be created. The threshold selection operator must then decrease the string lengths, such that only fit schemata remain. However, the threshold selection operator has proven quite unfit in this task [11].

3.2 Gene Expression Messy GA

Another messy genetic algorithm was developed by Kargupta [12]. The process of gene expression as observed in nature inspired his approach. Consequently this type of algorithm is called *gene expression messy genetic algorithm (GEMGA)*. The linkage learning is done by induction; the genes that improve the solution's pay-off are assumed to correlate. In a *first transcription phase* the contribution of a gene is to the fitness of the individual is determined. This is done by flipping each gene to its opposite value if the fitness increases, the original value does not contribute to the fitness, otherwise it does and is marked such that it cannot be changed in the future. In the *second transcription phase* all genes in a chromosome that have been marked as unchangeable are collected and compared with the same unchangeable genes of another randomly chosen chromosome. The intersection of the genes is saved (linkage set) and either is added to a list of the former chromosome or, if the set is already present, its weight is increased. After some iterations a matrix is build, which contains the probabilities of the presence of a gene under the condition that a specific gene is in the linkage set.

Afterwards, the schemata that have been identified as good are manifolded using *class selection*: two chromosomes are randomly picked, the fitter of both is marked, the genes in the linkage set of the marked chromosome are copied to the other chromosome, provided that the destroy genes exhibit less linkage than the genes by which they are replaced. Additionally tournament selection is applied.

Recombination is done by randomly picking an individual and selecting its maximum weighted linkage set, another individual is selected, and the corresponding genes are exchanged if the disrupted linkage sets in the latter chromosome have a smaller weight than the maximum weight of the former.

3.3 Linkage Learning GA

A completely different approach was taken by Harik [13]. Harik showed that a specific recombination operator, the so-called *exchange crossover operator*, could under certain conditions improve the linkage of genes. The chromosomes in Harik's linkage learning genetic algorithm (LLGA) are declared as rings. Each gene is described by its allele (value) and a locus, i.e., the interpretation position of this gene. By introducing so-called introns, genes that are not interpreted at all, the relative distance of two genes can be adjusted. In Figure 1 an example of a chromosome containing three genes is given. One can see that by inserting non-coding genes (introns) between the coding genes (exons) the distances (y_1 , y_2 , and y_3) can be varied.

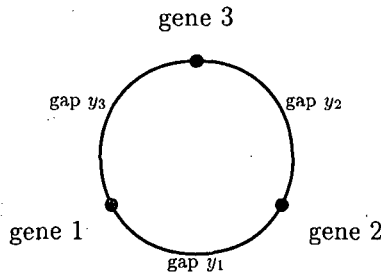


Figure 1: Chromosome in linkage learning genetic algorithm.

In contrast to most common implementations the *exchange crossover* operation is directional, that is, one individual serves as donor, the other one is the recipient. First some exchange material is randomly chosen from the donor chromosome, then a random graft point is declared at the recipient. The exchange material is then inserted within the graft point of the recipient. As one can see in Figure 2 the crossover leaves an over-determined chromosome, that is, some of the genes appear twice. Therefore an expression step is appended: A starting point and an interpretation direction are defined. Beginning from the starting point each gene that has been previously defined on the circle is simply removed, yielding a valid chromosome.

Harik proved that by applying this operator the individuals evolve towards tighter linkage. He assumed that the population will eventually consist mostly of both optimal building blocks and deceptive building blocks (as described earlier). This assumption can be made as the genetic algorithm eventually rules out all apparently unfit building blocks. Harik observed two effects:

- (1) *Linkage Skew*: tightly linked building blocks in the donor chromosome have a higher survival probability than loose linked building blocks. This mechanism is comparable to fitness-proportional selection.

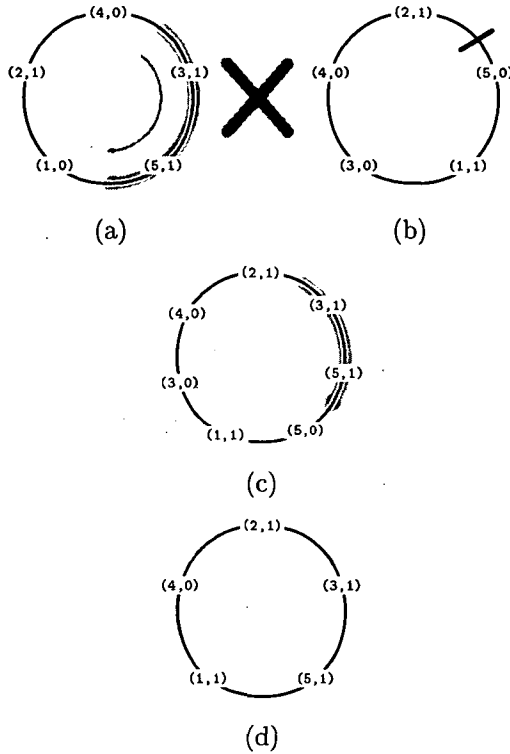


Figure 2: Exchange Crossover Operator: (a) donor, (b) recipient, (c) offspring before expression, and (d) offspring after expression.

- (2) *Linkage Shift*: if exchange material from the donor is copied onto the recipient—which contains an optimal building block—the building block is either disrupted or its linkage is increased.

However, in order for linkage learning to work selection must be slowed down, since it counteracts the evolution towards tighter linkage (as shown by Harik). Harik suggests two different methods to slow down selection:

- (1) *Restricted tournament selection*: In contrast to conventional selection operators where each individual is competing against one another, with tournament selection individuals only replace solutions which have a similar bit-string [13].

Thus, this selection operator is not only well suited for multi-modal optimization tasks, but will also maintain a high level of diversity within the population.

The main program's pseudo code listing in Section 4.5 comprises a detailed description of restricted tournament selection.

- (2) Probabilistic expression: This approach refers to an alternative way of chromosome encoding in which all genes appear twice, exhibiting the actual allele and its opposite. The starting point of the chromosome interpretation is randomly changed, resulting in a change of the genes' alleles with some probability. Thus, even if an allele has leveled out in the population, it might be revived.

3.4 Summary

The *messy genetic algorithm* and its variant the *fast messy genetic algorithm* can be considered early approaches. They have proven to work in limited settings, but haven't proven infeasible for real-world applications. A very promising approach has been suggested by Kargupta [12]. However, the number of additional fitness evaluation (in the transcription phase) and the large administration effort, which is necessary in order to store linkage information, seem to be a remarkable drawback. Harik's LLGA, on the other hand, proved to work well on exponentially scaled problems, that is, problems where parts of the genes contribute differently to the fitness value. As the hard problems for *GeLog* can be assumed to belong to this kind of problem class, this approach seems well suited for *GeLog*.

4 Improving the GeLog Framework

This section introduces the modifications that were applied to the data structures and the operators, which were implemented in order to achieve linkage learning. We have chosen the Linkage Learning GA (LLGA) approach to achieve this goal, since it offers a relatively good scalability and the genotypic representation is appropriate for *GeLog*. Moreover, the apparently reasonable theory of the LLGA and the promising results suggested an application to the *GeLog* framework.

The probabilistic expression (PE) as suggested by Harik [13] is not incorporated for the maintenance of diversity. Instead, tournament selection and restricted tournament selection are used. While tournament selection is a standard selection scheme in genetic algorithms, restricted tournament selection is commonly used for multi-objective optimization problems [13].

4.1 Chromosome

The genotypes in *GeLog* are represented by a so-called object graph [1], which allows for a direct transformation into the data structures used in logic programming. However, this representation is not ideal for the processing of building blocks in genetic algorithms. Not only is there evidence [5] that short alphabets have a positive impact on the implicit parallelism, but also for the linkage learning a chromosome of fixed length seems more appropriate. It is important that the entire search space is explicitly represented in one individual.

Except for the necessary changes in the genotype representation, the new version tried to stay as close to the original representation as possible. As in the

original work, each individual contains a number of disjunctions. However, in contrast to the previous implementation the number of disjunctions is fixed, i.e., each individual consists of the same number of disjunction. A gene, or bit, in the chromosome stands for one conjunction. A conjunction represents one variable assignment corresponding to the respective predicate.

For example, let the background knowledge, i.e., the pool of valid predicates which may be used at the right hand side of the individual, be:

female/1 parents/3

with the target predicate: daughter(X0, X1)

In the original work an individual could look as follows:

daughter(X0, X1) :- female(X0), parents(X1, X0, X1).

parents(X0, X1, X1).

female(X1), female(X0), parents(X1, X1, X0).

The new representation consists of a fixed number of fixed length bit strings. The individual must hence be transformed into something like this:

1000000100	(1. disjunction)
0000000010	(2. disjunction)
1100010000	(3. disjunction)

How do we achieve an appropriate representation?

First of all, we have to determine the length of the chromosome, since it will be fixed throughout the entire process. Thus, the chromosome's length must allow for encoding all valid predicates with all possible assignments:

$$l = \sum_{p \in B} \text{arity}(t)^{\text{arity}(p)},$$

where B is the background knowledge, p is one of the background knowledge's predicates, and t is the target predicate.

For the former example the length would be:

$$\begin{aligned} l &= \text{arity}(\text{daughter})^{\text{arity}(\text{female})} + \text{arity}(\text{daughter})^{\text{arity}(\text{parents})} \\ &= 2^1 + 2^3 = 10 \end{aligned}$$

If we introduce a number of additional, unbound variables (v) that the literals may take as arguments this can be transformed into

$$l = \sum_{p \in B} (\text{arity}(t) + v)^{\text{arity}(p)}.$$

In our example we allow for additional two variables, with the target predicate's original two variables we obtain variables $X0..X3$:

$$l = (arity(daughter) + v)^{arity(female)} + (arity(daughter) + v)^{arity(parents)} \\ = 4^1 + 4^3 = 68$$

Let $p_n(x)$ be the n th predicate of the background knowledge with allocation $x = (x_0, \dots, x_m)$; where $m = arity(p_n) - 1$. All variables x_i (with $0 \leq i \leq m$) must be one of the target predicate's or of additional variables: $0 \leq x_i < (arity(t) + v)$. The locus of this mapping can be calculated as follows:

$$locus(p_n(x)) = \sum_{j=0}^{n-1} (arity(t) + v)^{arity(p_j)} + \sum_{i=0}^{arity(p_n)-1} x_i \cdot (arity(t) + v)^i.$$

Let us calculate the locus of the parent/3 predicate of the former example using allocation $parents(X3, X1, X1)$.

$$locus(parents(X3, X1, X1)) = (arity(daughter) + v)^{arity(female)} \\ + 3 \cdot (arity(daughter) + v)^0 + 1 \cdot (arity(daughter) + v)^1 \\ + 1 \cdot (arity(daughter) + v)^2 = 4^1 + 3 + 1 \cdot 4 + 1 \cdot 16 = 27$$

Thus, gene number 27 indicates whether the predicate $parents(X3, X1, X1)$ is present or not. Its allele (value) is either 1 or 0.

The coding of the genes is messy, that is, their position in the bit string is not fixed but they may float around. A gene's predicate allocation is not determined by the gene's position in the bit string but by its locus, which is in general different from the position.

It is quite obvious that the length of the chromosomes is increasing exponentially with the arity of the predicates of background knowledge and the arity of the target predicate. This is problematic since the genes have messy coding, which means every single gene contains a number as large as the chromosome's length. This is necessary since the locus—the position of the genes within the chromosome—has to be stored. For example, for 100 literals in the background knowledge, an average arity of 10, and a target predicate's arity of 10 the chromosome length is 10^{12} , a number that must be stored in all 10^{12} genes of the chromosome.

4.2 Mutation

With the new representation the change of one single bit deletes or adds one mapping of a predicate. Thus, a single allocation, a predicate, or even a disjunction may be erased altogether by the change of one bit. The following example illustrates this: Let the background knowledge be the same as in the previous example. The three rows represent the variable numbers within the literal. The columns denote the subscript of the X variables. There are four variables ($X0, X1$ are arguments of

the target literal daughter; X2, X3 are unbound variables), and we need to be able to place any of these four variables on any position within each literal. Therefore, we only need four positions for the female literal, as it takes only one argument (i.e., X0, X1, X2, or X3). For the second literal, parents, we need 4^3 positions, since it requires three arguments.

1. Variable	0123	0123	0123	0123	0123	0123	0123	0123	0123	0123	0123	0123	0123	0123	0123	0123	0123	0123
2. Variable		0000	1111	2222	3333	0000	1111	2222	3333	0000	1111	2222	3333	0000	1111	2222	3333	3333
3. Variable		0000	0000	0000	0000	1111	1111	1111	1111	2222	2222	2222	2222	2222	3333	3333	3333	3333

0010|0000|0100|0000|0100|0000|0000|0000|0100|0000|0000|0010|0000|0000|0000|0000|0000

is equivalent to the structure:

female(X2), parents(X1, X1, X0), parents(X1, X3, X0),
parents(X1, X3, X1), parents(X2, X2, X2).

Two mutations are performed:

$\begin{array}{c} \downarrow \qquad \qquad \qquad \downarrow \\ \Rightarrow \quad 0010|0000|0100|0000|0100|0000|0000|0000|0100|0000|0000|0010|0000|0000|0000|0000|0000 \\ \quad \quad 0010|0000|0100|0010|0100|0000|0000|0000|0100|0000|0000|0000|0000|0000|0000|0000|0000 \end{array}$

This is equivalent to inserting a new predicate allocation (parents(X2, X2, X0)) and deleting one (parents(X2, X2, X2)):

female(X2), parents(X1, X1, X0), parents(X2, X2, X0),
parents(X1, X3, X0), parents(X1, X3, X1).

Since the modified version of *GeLog* aims at maintaining a high level of diversity, it does not depend on mutation. Compared to the original work, the mutation rates have therefore been decreased drastically.

4.3 Diversity

In order to make linkage learning work, it is necessary to maintain a large diversity in the population. It might therefore be desirable to keep solutions in different parts of the search space and optimize these solutions individually. A better solution replaces another solution only if both are similar to another, i.e., if their distance is small.

In order to evaluate the distance of two individuals, their chromosomes have to be compared. Since the chromosomes are bit strings, the Hamming distance (i.e., the sum of all difference bits) is an appropriate distance metric. As individuals in *GeLog* do not only consist of one but a number of chromosomes, the Hamming distances for all chromosomes have to be evaluated. One approach is to calculate the Hamming distance for each pair of chromosomes of the two individuals (x and y), after which the sum of these chromosome-wise distances yields the distances of the individuals:

$$\sum_{c=0}^m \sum_{i=0}^n |x_i^c - y_i^c|,$$

where m is the number of chromosomes and n is the number of genes. In contrast to a single chromosome where a gene at a specific locus always encodes the same trait, it cannot be pre-terminated what kind of disjunction one chromosome will be coding for. Therefore, the distance between two individuals is not as obvious as in a single chromosome case. One extreme case is, two individuals containing the same chromosomes yet in a different order.

One solution to this problem is to form the sum of the distances of all chromosomes in one solution with all the chromosomes in the other solution:

$$\sum_{c=0}^m \sum_{d=0}^m \sum_{i=0}^n |x_i^c - y_i^d|,$$

where m is the number of chromosomes and n is the number of genes. This distance metric is referred to as *sum/sum*.

This approach, however, does not aim at finding corresponding chromosomes in the compared individuals. For example, although the sum of all distances might be large, the distance between certain chromosomes is possibly small. The *sum/min* approach takes this into account by identifying matching slots. By finding the permutation p of disjunctions that maximizes

$$\sum_{c=1}^m \frac{\min_{\substack{1 \leq j \leq m; \\ j \neq c}} \left\{ \sum_{i=0}^n |x_i^c - y_i^{p_j}| \right\}}{\sum_{i=1}^n |x_i^c - y_i^{p_c}|}$$

those disjunctions are identified that exhibit a much smaller distance than the disjunctions with the second smallest distance.

```

for all  $c_1 :=$  chromosomes in individual1 do
  find  $c_2 :=$  unmarked chromosomes in individual2 with minimum distance to  $c_1$ 
  if two chromosomes have the same minimum distance then
    choose one randomly
  end if
  store distance
  mark chromosome  $c_2$ 
end for
sum up all stored distances

```

Figure 3: Pseudo code of the *sum/min* algorithm

The algorithm's complexity is $O(n \cdot n!)$ in the number of chromosomes. Since distance comparisons are needed very frequently, *GeLog* uses a variation of this procedure. Instead, all distances between all chromosomes are evaluated and those chromosomes are assumed to match that have the smallest distance. If any chromosome has the same distance to more than one chromosome in the other individual, one of these chromosomes is chosen randomly. Figure 3 demonstrates the algorithm.

4.4 Recombination

The different recombination operators of the original *GeLog* version, as explained in Section 2, have been replaced by a single operator. Selection yields two individuals, a donor and a recipient. First one chromosome is selected in the donor individual and a corresponding chromosome in the recipient is chosen. After crossover has been performed a slot for the new recombined chromosome has to be chosen. This process of choosing chromosomes, recombining them and storing them is repeated for all chromosomes.

Since it is not obvious how to select chromosomes for recombination, several strategies have been developed:

- (1) **ordered**: the chromosomes are selected in the order they appear in the individual,
- (2) **shuffled**: the chromosomes are randomly shuffled and selected in this new order, or
- (3) **fitness**: the chromosomes are selected fitness proportionally, i.e., by roulette wheel selection. This scheme is very expensive, as a huge number of fitness evaluations have to be performed.

The new individual (offspring) is now created by the recombined chromosomes. However, each chromosome has to go into a different slot in the offspring. Therefore, a selection strategy is also required for storing:

- (1) **ordered**: the chromosome is stored in the same slot as the recipient's chromosome was selected from,
- (2) **shuffled**: a randomly shuffled list of all chromosomes slots is generated, the chromosomes are stored in that new order, or
- (3) **similarity**: the chromosome is placed into the slot that has the shortest distance, i.e., the number of the recipients slot which is most similar to the recombined chromosome.

The most important of the nine possible combinations are explained in the following:

- (1) **ordered/ordered**: Each chromosome has its fixed slot, for the whole evolutionary process.
- (2) **shuffled/similarity**: The parents are chosen randomly, the offspring, however, replaces the chromosome, which it is most similar to. This selection/storing scheme induces a similar distribution on the single individual as the restricted tournament selection did on the entire population.
- (3) **shuffled/shuffled**: The parents are selected randomly, the offspring is stored at a random position. This scheme is suitable if restricted tournament selection is used and all clauses should be intermixed.

- (4) **fitness/similarity**: As shuffled/similarity, but holds the danger that the inner-individual diversity gets lost too fast since only well performing chromosomes are selected.

As introduced by Harik [13] an *exchange crossover operator* was used to recombine two chromosomes. When combined with a harsh control of selection, this operator induces a tighter linkage on the building blocks.

4.5 Flow of GeLog

In this section, pseudo-code is presented for the fitness evaluation routine and for the main program.

The fitness evaluation consists of two parts: first the genome of the individual has to be decoded into a hypothesis, and in a second step it is checked, whether the hypothesis correctly classifies all training instances. The hypothesis must classify negative instances as false and positive instances as true. The fitness value corresponds to the percentage of correctly classified training instances.

Fitness Evaluation

```

input individual A
hypothesis  $H := \text{decode } A$ 
for all positive training instances  $EP[j]$  ( $j := 0.. \text{number of pos. instances}$ ) do
    if  $H$  accepts  $EP[j]$  then
        increase fitness of  $A$ 
    else
        decrease fitness of  $A$ 
    end if
end for
for all negative training instances  $EN[j]$  ( $j := 0.. \text{number of neg. instances}$ ) do
    if  $H$  rejects  $EN[j]$  then
        increase fitness of  $A$ 
    else
        decrease fitness of  $A$ 
    end if
end for

```

The following flow demonstrates that the GA flow differs substantially depending on the selection operator. With conventional selection operators the fitness of individuals is evaluated after the new population has been created, whereas with restricted tournament selection more evaluation steps have to be performed.

The program has been implemented using the programming language C++; all experimental runs have been conducted on Intel/AMD processor based computers running the Linux operating system. As PROLOG interpreting system the *SICStus* framework version 3.8.5 was used, which can be easily linked to C/C++ programs. However, calling the external PROLOG process is expensive and decidedly contributes to the time required by fitness evaluations. This circumstance is

especially problematic with restricted tournament selection, for which a number of additional fitness evaluations have to be performed.

Main Program

```

initialize prolog interpreter
load background knowledge
randomly initialize start population
for all individuals  $A[i]$  ( $i := 0..population\ size$ ) do
    evaluate  $A[i]$ 
end for
while not (termination criterion reached or max. number of generations) do
    if selection = restricted tournament then
        while not new population complete do
             $A :=$  select randomly
             $B :=$  select randomly
             $A' :=$  exchange crossover  $A, B$ 
             $B' :=$  exchange crossover  $B, A$ 
            mutate  $A', B'$ 
             $W[] :=$  select randomly  $w$  individuals
             $A'' := W[j]$ , where  $distance(W[j], A') = \min_{i:=1..w} (distance(W[i], A'))$ 
            evaluate  $A'$ 
            if fitness  $A' >$  fitness  $A''$  then
                replace  $A''$  with  $A'$ 
            end if
            repeat the same for  $B'$ 
        end while
    else
        while not new population complete do
             $A :=$  select individual (using any recombination operator)
             $B :=$  select individual (using any recombination operator)
             $A' :=$  exchange crossover  $A, B$ 
             $B' :=$  exchange crossover  $B, A$ 
            mutate  $A', B'$ 
            place  $A', B'$  into the new population
        end while
        evaluate all individuals in the new population
    end if
end while

```

5 Experimental Results

Two experiments that had proven difficult for the original version of *GeLog* have been conducted in order to verify that the performance of *GeLog* has been improved.

5.1 Tic-Tac-Toe

This experiment was introduced by Aha [14] and is based on the Tic-Tac-Toe game. It encodes all possible endgame situations where the player using the “X” symbol has started. The target concept, which takes the nine board squares as variables with values blank, X, or O, is to classify the win situation for player “X”. For 626 of the 956 possible constellations player “X” wins.

The performance of the concept learning algorithms on this data set varied remarkably, depending on the used variant of learners: while experiments conducted with decision tree based learners exhibited errors of 20% or more, other algorithms, such as rule based learners performed well on it (errors $\leq 2\%$). With the original version of *GeLog* we could not achieve error results below 24%, which were significantly improved after the modifications.

individual distance	chromosome selection	chromosome storing	selection operator	lowest error (%)	average error (%)
none	ordered	ordered	ts	6.25	9.71
none	shuffled	similarity	ts	5.26	8.66
sum/min	shuffled	similarity	rts	6.25	10.13
sum/sum	shuffled	similarity	rts	6.25	10.86
original <i>GeLog</i> version				24.95	26.91

Table 1: GeLog test results on ‘TicTacToe’

Table 1 shows four tests, each consisting of five test runs. All tests have been conducted using ten-cross-validation, that is, the example set is divided into 10 disjunctive subsets. One is declared as test set. This procedure is used in order to avoid over-fitting. The first column indicates the individuals’ distance metric. For tournament selection this column contains the word “none”, since this selection operator does not utilize the individuals’ distances. The second and third columns state the chromosome selection and storing scheme as described in Section 4.4. The selection operator appears in the fourth row, “rts” means restricted tournament selection, whereas “ts” is tournament selection. The depicted settings have been chosen in order to clarify whether restricted tournament selection is in any case necessary to maintain a high level of diversity. Therefore, tournament selection was tested using the ordered selection and storing scheme, thus totally ignoring the distance relation. The second experiment was conducted using random selection but with storage into the slot that exhibits the highest similarity. For restricted tournament experiments the same selection/storing scheme was used with the two different distance metrics, as this might be a critical factor for restricted tournament selection. The mutation probability for all experiments was defined as 0.01, crossover was performed with a probability of 0.6, a population consists of 150 individuals, and the maximum number of generations was 250, however, for tournament selection, the lowest error rate was chiefly obtained after 50–100 generations.

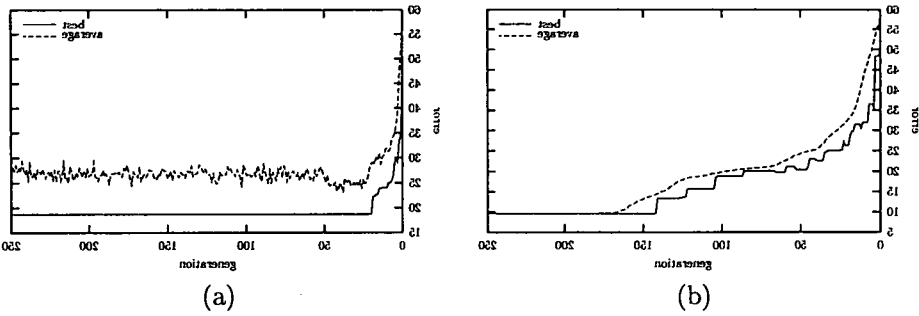


Figure 4: Minimal versus average error for “TicTacToe”: (a) tournament selection, (b) restricted tournament selection

The absence of a significant difference in the results of tournament and restricted tournament selection may indicate that the problem is not difficult enough, i.e., that plain tournament restriction can maintain large enough a diversity.

For the chromosome selection/storing we see a slight difference between “ordered/ordered” and “shuffled/similarity”, which can be attributed to higher in-individual diversity, that is, clauses have greater differences.

Figure 4 shows two runs of the same experiment, however only 100 individuals per population are generated. It becomes clear that restricted tournament selection is aiming at improving the fitness situation of the entire population.

individual distance	selection operator	run time (25 generations)
none	ts	≈ 3m50s
sum/sum	rts	≈ 5m
sum/min	rts	≈ 5m8s
original <i>GeLog</i> version		≈ 2m10s

Table 2: Durations for different selection operators and distance metrics for 25 generations.

In Table 2 the durations for different operators and distance metrics are demonstrated. The original version of *GeLog* is faster, due to faster decoding times and less fitness evaluations. It is also obvious that restricted tournament selection has a significantly longer run time, owing to the higher number of fitness evaluations. The different distance metrics seem to have little influence on the duration.

All experiments have been executed on an Intel Pentium II computer with 450MHz. The run time experiments only involved 25 generations. The number of right hand sides (disjunctions) was fixed to three. The length of a chromosome is 27.

5.2 Chess Endgame King-Rook-King

The second experiment we have chosen to validate the performance improvement of *GeLog* is a chess endgame variant, the “White King and Rook vs. Black King” [15]. There are three pieces left on the board: a white king, a white rook, and a black king. The next player to move is white. The objective is to classify legal and illegal constellations, where a situation is illegal when either white has already won or black can capture the rook without being check (draw).

individual distance	chromosome selection	chromosome storing	selection operator	lowest error (%)	average error (%)
none	ordered	ordered	ts	0.53	0.71
sum/sum	ordered	ordered	rts	0.53	0.71
sum/sum	shuffled	shuffled	rts	0.53	0.72
sum/min	shuffled	shuffled	rts	0.53	0.72
sum/sum	shuffled	similarity	rts	0.53	0.72
sum/min	shuffled	similarity	rts	0.53	0.74
original <i>GeLog</i> version				4.90	8.02

Table 3: GeLog test results on ‘King-Rook-King’

There is a total of 28056 entries, each of which consists of the coordinates of the pieces and an attribute for the optimal number of moves for white to win. The attributes are the number of moves (0..17) and “draw”.

mutation probability	0.01
crossover probability	0.6
population size	150
number of generations	250
termination criterion	–

Table 4: Parameter Settings for the ‘King-Rook-King’ experiment

With the original *GeLog* program the minimum error was about 5%. Table 3 shows the experimental results for the modified version of *GeLog*, each entry representing five test runs, all of which are conducted using ten-cross-validation. The experiment settings are summarized in Table 4.

Again, neither the different distance metrics nor the diversity sustaining restricted tournament selection exhibit a remarkable difference with respect to the objective function values.

Also Figure 5 shows that the average payoff of the population is remarkably higher for restricted tournament selection.

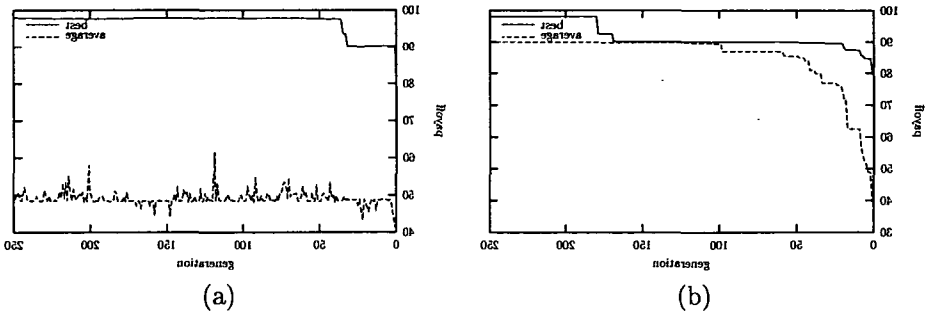


Figure 5: Best versus average payoff for “King Rook King”: (a) tournament selection, (b) restricted tournament selection

In Table 5 the run times for the different operators and metrics are listed. The experiments have been conducted with the same settings as in the previous experiment. However, the number of disjunctions is set to five. The length of chromosome in the new version is 54.

individual distance	selection operator	run time (25 generations)
none	ts	≈ 3m50s
sum/sum	rts	≈ 4m50s
sum/min	rts	≈ 5m10s
original <i>GeLog</i> version		≈ 2m6s

Table 5: Run times for different selection operators and distance metrics for 25 generations.

6 Conclusion and Future Work

In this article we presented some modifications to the *GeLog* framework—a genetic logic programming system—in order to improve its underlying genetic algorithm. The original *GeLog* program, linkage learning, and some related approaches were briefly introduced.

It was demonstrated how linkage learning was incorporated into the *GeLog* framework. All necessary changes and the resulting problems were presented. A distance metric, which was developed for the implemented selection operator, was also presented.

Finally, the test results showed that the *GeLog* framework has been significantly improved. Problems that used to be hard for the original program were solved.

Although these first test results are very promising and already demonstrate that the modifications do indeed enhance *GeLog*'s performance, further experiments have to be conducted to quantify the influence of the improvements achieved by the modified *GeLog* framework. In particular, it has to be investigated if under certain conditions linkage learning by the combination of the exchange crossover operator and standard tournament selection can withstand the force of selection.

References

- [1] Gabriella Kókai. *GeLog—A System Combining Genetic Algorithm with Inductive Logic Programming*. In *Proc of the International Conference on Computational Intelligence, 7th Fuzzy Days LNCS*, pages 326–345, Springer Verlag, Dortmund, 2001.
- [2] Nada Lavrač and Sašo Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York, 1994.
- [3] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.
- [4] Christian Jacob. *Illustrating Evolutionary Computation with Mathematica*. Morgan Kaufmann, San Francisco, CA, 2001.
- [5] John H. Holland. *Adaptation in Natural and Artificial Systems*. PhD thesis, University of Michigan, Ann Arbor, 1975.
- [6] John J. Grefenstette. Deception considered harmful. In D. Whitley, editor, *Proceedings of the Foundations of Genetic Algorithms Workshop*, Morgan Kauffmann, Vail, CO, 1992.
- [7] Stephanie Forrest and Melanie Mitchell. What Makes a Problem Hard for a Genetic Algorithm? Some Anomalous Results and Their Explanation. *Machine Learning*, 13:285–319, 1993.
- [8] David E. Goldberg and Clayton L. Bridges. An analysis of a reordering operator on a GA-hard problem. *Biological Cybernetics*, 62(5):397–405, 1990.
- [9] David E. Goldberg, Bradley Korb, and Kalyanmoy Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530, 1990.
- [10] David E. Goldberg, Kalyanmoy Deb, Hillol Kargupta, and Georges Harik. Rapid accurate optimization of difficult problems using fast messy genetic algorithms. In Stephanie Forrest, editor, *Proc. of the Fifth Int. Conf. on Genetic Algorithms*, pages 56–64, Morgan Kaufmann, San Mateo, CA, 1993.
- [11] Hillol Kargupta. SEARCH, polynomial complexity, and the fast messy genetic algorithm. Technical report, University of Illinois, Illinois Genetic Algorithms Laboratory, Urbana, IL, 1995.

- [12] Hillol Kargupta. The gene expression messy genetic algorithm. In *International Conference on Evolutionary Computation*, pages 814–819, Piscataway, NJ, 1996.
- [13] George Harik. *Learning linkage to efficiently solve problems of bounded difficulty using genetic algorithms*. PhD thesis, The University of Michigan, Ann Arbor, Michigan, 1997.
- [14] David W. Aha. Incremental constructive induction: An instance-based approach. In *Proceedings of the 8th International Workshop on Machine Learning*, pages 117–121, Morgan Kaufmann, Evanston, IL, 1991.
- [15] Micheal Bain. Learning optimal KRK strategies. In S. Muggleton, editor, *ILP92: Proc. Intl. Workshop on Inductive Logic Programming*, Report ICOT TM-1182, Tokyo, 1992.

Various Robust Search Methods in a Hungarian Speech Recognition System*

Gábor Gosztolya[†] András Kocsor[‡] László Tóth[‡]
and László Felföldi[‡]

Abstract

This work focuses on the search aspect of speech recognition. We describe some standard algorithms such as stack decoding, multi-stack decoding, the Viterbi beam search and an A* heuristic, then present improvements on these search methods. Finally we compare the performance of each algorithm, grading them according to their performance. We will show that our improvements can outperform the standard methods.

KeyWords. search methods, stack decoding, multi-stack decoding, Viterbi beam search.

1 Introduction

In any speech recognition system, the real task is to find the most probable word (sequence of phonemes) for a given speech signal. However, as the number of possibilities is extremely high, and most of them will have very low probabilities, we need efficient algorithms to reduce the enormous search space. There are numerous standard methods for doing this, and some rarely used heuristics. We implemented and tested some of them, and adapted these according to our needs. Our aim was to construct a faster method which recognized the same amount of words. The methods were tested within the framework of our segment-based recognition system, the OASIS Speech Laboratory [7, 8].

*This work was supported under the contract IKTA No. 2001/055 from the Hungarian Ministry of Education.

[†]Research Group on Artificial Intelligence of the Hungarian Academy of Sciences and University of Szeged, H-6720 Szeged, Aradi vértanúk tere 1., Hungary,
e-mail: ghosty@rgai.inf.u-szeged.hu, kocsor@inf.u-szeged.hu, tothl@inf.u-szeged.hu

[‡]Department of Informatics, University of Szeged H-6720 Szeged, Arpád tér 2., Hungary,
e-mail: lfelfold@inf.u-szeged.hu

2 A Segment-based Speech Recognition Approach

In the following, the speech signal A will be treated as a chronologically increasing series of the form $a_1 a_2 \dots a_{t_{max}}$, while the set of possible phoneme-sequences will be denoted by W . Essentially the task here is to find the word $\hat{w} \in W$ defined by

$$\hat{w} = \arg \max_{w \in W} P(w|A) = \arg \max_{w \in W} \frac{P(A|w) \cdot P(w)}{P(A)} = \arg \max_{w \in W} P(A|w) \cdot P(w),$$

where $P(w)$ is known as the *language model*.

If we optimize $P(w|A)$ directly, we use a discriminative method, while if we use Bayes' theorem and omit $P(A)$, our approach is generative. Moreover the recognition process can be frame-based or segment-based, depending on whether the model incorporates frame-based or segment-based features. The widely-used HMM is a frame-based, generative method, but in the following we will describe the recognition process in a segment-based, discriminative approach (c.f. [8]).

We assume that $P(w|A) = \prod_i P(w_i|A) = \prod_i P(w_i|A_i)$, i.e. that the phonemes are independent, and for a word $w = o_1 \dots o_l$ a phoneme o_i is based on $A_i = a_j a_{j+1} \dots a_{j+r-1}$ (an r -long segment of A , where $A = A_1 \dots A_n$). With this A_i segment, a *phoneme classifier* identifies the phoneme by some method using long-term features; in our recognition system, the OASIS Speech Laboratory, Artificial Neural Networks (ANNs) [3] are used, but the way the classifier actually works is of no concern to us here.

To determine the values of these $P(w_i|A_i)$ functions, we need to know the exact values of the A_i s, which are determined by their start and ending times (the above j and $j + r - 1$ values). Alas, this is quite a hard task, and because automated segmentation cannot be done reliably, the program will make many segmentation hypotheses, so we must include this segmentation T in our formulae:

$$P(w|A) = \sum_T P(w, T|A) = \sum_T P(w|T, A) \cdot P(T|A) \approx \max_T P(w|T, A) \cdot P(T|A)$$

For a given T , $P(w|T, A)$ can be readily calculated with the phoneme classifier, and we handle $P(T|A)$ using ANN-s as well. For this two-class training, the elements of the "phoneme" class were marked by hand, while the others in the "anti-phoneme" class were constructed from randomly selected parts of two or more phonemes. This allows us to employ the same set of features in the segmentation procedure that was used for phoneme recognition.

3 Overview of Robust Search Methods

3.1 Definition of the search space

Before presenting the algorithms, we have to define some basic terms and notations. An array $T_n = [t_0, t_1, \dots, t_n]$ is called a *segmentation* if $0 = t_0 < t_1 < \dots < t_n \leq$

t_{max} holds, i.e. they are in increasing chronological order. We also require that every phoneme fit into some overlapping interval $[t_i, t_j]$ ($i, j \in \{0, \dots, n\}, 0 \leq i < j \leq n$), i.e. the former speech segments are referred by their start and end times.

Given a set of words W , $Pref_k(W)$ will denote the k -long prefixes of all the words in W with at least k phonemes. Then we may construct the search tree in a recursive manner: $h_0 = (\emptyset, [t_0])$ will be the root of the tree, and $Pref_1(W) \times T_n^1$ will contain the first-level vertices. Then, for a $(o_1 o_2 \dots o_j, [t_{i_0}, \dots, t_{i_j}])$ leaf we link all $(o_1 o_2 \dots o_j o_{j+1}, [t_{i_0}, \dots, t_{i_j}, t_{i_{j+1}}]) \in Pref_{j+1}(W) \times T_n^{j+1}$ nodes.

When one or more hypothesis is discarded due to its high cost, we say that it was *pruned*.

For the algorithms, certain notations are employed. " \leftarrow " means that a variable is assigned a value; " \Leftarrow " means pushing a hypothesis into a stack. A $H(t, c, w)$ hypothesis is a triplet of time, cost and a phoneme-sequence. *Extending* a hypothesis $H(t, c, w)$ with a phoneme v and an ending time t_i results in a hypothesis $H'(t', c', w')$, where $t' = t_i$, $w' = wv$, and $c' = c + c_i$, c_i being the cost of v in an interval $[t, t_i]$. This is equivalent to $p' = p \cdot p_i$, where p' , p and p_i are the probabilities of H' , H , and v in an interval $[t, t_i]$, respectively, and $c_i = -\ln p_i$. We are looking for the hypothesis with the lowest cost.

3.2 Stack decoding

The stack decoding algorithm [2] is time-asynchronous, i.e. it compares hypotheses with different ending times.

In the first step of the process we place the initial hypothesis into the stack. Then we pop the hypothesis in order to examine and extend it. Next, we put all the new hypotheses into the stack and pop the most probable of them. We repeat the process until the popped hypothesis reaches the end of the utterance.

The above algorithm works because it extends hypotheses, and their cost increases since we add these costs (non-negative real numbers) together. Thus, when we reach the end of the utterance, all unexamined hypotheses will have higher costs than our actual solution.

In practice it is common to use a finite stack. However, for large vocabularies and/or sentences (those with a huge search space) there is a danger that it will eliminate the best scoring hypotheses with a greater end time. Another problem with this method is that, by increasing the length of an utterance, the run time of the stack decoding algorithm will increase exponentially.

3.3 An A* heuristic

The A* search [4] algorithm is also a common method for finding a near-optimal solution. Here, besides the $g(H)$ value for a hypothesis H (the cost so far), there is a $h(H)$ value for estimating the cost of the remaining path. We put the hypotheses into a stack and sort them using $f(H) = g(H) + h(H)$. Basically, this is just a variation of the *stack decoding* method.

Algorithm 1 Stack decoding algorithm

```

Stack  $\leftarrow H_0(t_0, 0, \emptyset)$ 
while Stack is not empty do
   $H(t_i, p, w) \leftarrow \text{top}(\text{Stack})$ 
  if  $t_i = t_{\max}$  then
    return H
  end if
  for  $t_l = t_{i+1} \dots t_{\max}$  do
    for all  $\{v \mid wv \in \text{Pref}_{1+\text{length of } w}\}$  do
       $H'(t_l, p', w') \leftarrow \text{extend } H \text{ with } v \text{ on } [t_i, t_l]$ 
      Stack  $\leftarrow H'$ 
    end for
  end for
end while

```

Jelinek offers a method for constructing a heuristic based on examples. The exact formulae can be found in [6]. The idea behind it is simple enough. An evaluation is made on segmented, tagged data in order to calculate the average (or the minimum) cost per unit time. It should then give a good estimate of the cost for the remaining time. As regards the optimality criterion, the estimate must be not greater than the actual cost. It is quite hard to meet this criterion using the average-value based approach, but fairly straightforward to satisfy with the latter. However, when we calculate the minimum cost per unit time using the latter version, there is a certain loss of efficiency although it is still somewhat better than the simple stack decoding method. The solution might be to use some hybrid combination of the two.

Algorithm 2 Universal A* algorithm

```

Stack  $\leftarrow H_0(t_0, 0, \emptyset)$ 
while Stack is not empty do
   $H(t_i, p, h, w) \leftarrow \text{top}(\text{Stack})$ 
  if  $t_i = t_{\max}$  then
    return H
  end if
  for  $t_l = t_{i+1} \dots t_{\max}$  do
    for all  $\{v \mid wv \in \text{Pref}_{1+\text{length of } w}\}$  do
       $H'(t_l, p', h', w') \leftarrow \text{extend } H \text{ with } v \text{ on } [t_i, t_l]$ 
      Stack  $\leftarrow H'$ 
    end for
  end for
end while

```

3.4 Multi-stack decoding

This method is a time-synchronous modification of stack decoding. Instead of using just one stack (where the elements cannot truly be compared because most of them have different end times), we assign one stack for each time instance. Advancing in time, we can pop each hypothesis one at a time from the given stack, extend them, and put the new hypotheses into the right stack (which depends on their new end time) [1].

Obviously stack size is very important in this method as it can affect accuracy. Overly large stacks result in a large search space (and unnecessarily long run time), while very small stacks can prune those hypotheses whose extensions might be better at a later time. Note that, for a given stack size, the run time of the algorithm depends only on the length of the utterance (or, to be more precise, on the number of possible segments).

Algorithm 3 Multi-stack decoding algorithm

```

Stack[t0] ← H0(t0, 0, ∅)
for time = t0 ... tmax do
  while not empty(Stack[time]) do
    H(t, p, w) ← top(Stack[time])
    if time = tmax then
      return H
    end if
    for ti = time + 1 ... tmax do
      for all {v | wv ∈ Pref1+length of w} do
        H'(ti, p', w') ← extend H with v on [ti, ti]
        Stack[ti] ← H'
      end for
    end for
  end while
end for

```

3.5 Viterbi beam search

The standard Viterbi search algorithm is just the standard time-synchronous exhaustive search method but, as it stands, it is practically unusable. However, with a small modification it can be made rather effective. We employ a variable T called *beam width*; for each time instance t we calculate D_{min} , i.e. the lowest cost of the hypotheses with the end time t , and prune all those hypotheses whose cost D falls outside $D_{min} + T$ [5]. The value of the beam width is found by trial and error.

Several versions of this method exist. When choosing one we might use different beam widths for different end times (using greater values at the beginning of words). Or we could calculate the beam width dynamically (i.e. keeping the best N hypotheses – which is identical to the multi-stack decoding algorithm –, or

reducing the beam width when the probabilities start to decrease). In trials so far we have tested this method only with a constant beam width.

Algorithm 4 Viterbi beam search algorithm

```

Stack[t0] ← H0(t0, 0, ∅)
for time = t0 ... tmax do
  while not empty(Stack[time]) do
    H(t, p, w) ← top(Stack[time])
    if time = tmax then
      return H
    end if
    for ti = time + 1 ... tmax do
      for all {v | wv ∈ Pref1+length of w} do
        H'(ti, p', w') ← extend H with v on [time, ti]
        Stack[ti] ← H'
        Prune Stack[ti] with beam width T
      end for
    end for
  end while
end for

```

4 Refinement of the Multi-stack decoding algorithm

When calculating the optimal stack size for multi-stack decoding, it is readily seen that this optimum will be the one with the smallest value where no best-scoring hypothesis is discarded. But this approach obviously has one major drawback. Most of the time bad scoring hypotheses will have to be evaluated owing to the constant stack size. If we could only find a way of estimating the required stack size at each time instance, the performance of the method would markedly improve.

One possibility might be to combine multi-stack decoding with a Viterbi beam search. At each time point we keep the n best-scoring hypotheses, and discard those which are not close to the peak (thus the cost will be higher than the best cost plus the beam width). Here the beam width can also be determined empirically.

One surprising thing is that when we determine the optimal parameters (stack size and beam width) for the two methods (multi-stack and Viterbi beam), both parameters can be used together, thus making the combined search method work faster than either of them separately. We found that this worked for both test sets.

Yet another approach for improving the multi-stack method is that we can predict, at a given time instance, what stack size should be sufficient. We devised two improved methods based on this.

We trained an ANN to predict whether, at a given time instance, a bound between phonemes exists or not. Then, at each time instance, this ANN returns a

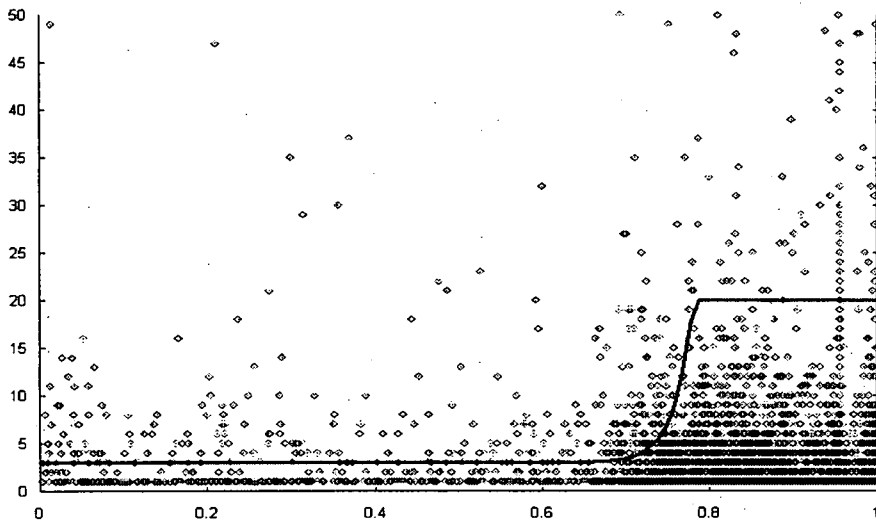


Figure 1: Bound probability – stack size diagram with the best fitting curve

probability p for this. In the first improvement we compare this p to a parameter l : if $p < l$, we use a smaller-sized stack (c_{min}), and a bigger (c_{max}) one otherwise.

We could also improve the model by fine-tuning it. To find a function that approximates the necessary stack size based on the output p of the ANN, we conducted an experiment. We recognized a set of test words using a standard multi-stack decoding algorithm with a large stack size. Then we examined the path which led to the winning hypothesis (or the first n hypotheses), and noted the required stack size and the phoneme-bound probability p at each time instance. The points of Figure 1 show the necessary stack sizes as a function of p .

For a phoneme-bound probability p (supplied by the ANN), we found that a $\min(c_0 + e^{c_1 \cdot p + c_2}, c_3)$ size stack was satisfactory. Obviously, the value for c_3 comes from the test of multi-stack decoding, and the value for c_0 from an examination of the previous improvement (as c_{min}). After, for a given c_1 , c_2 can be determined by trial and error. The best fitting curve was plotted in Figure 1.

5 Experimental results

5.1 The testing sets

In trials we tested the above methods and their variations using varying parameters, namely different dictionary sizes, words, and other parameters which are method dependent (e.g. stack size in stack decoding). We also examined whether making use of a voicedetect function (which seeks to remove long, silent parts of a voice

signal) significantly improves the speed of recognition, thereby reducing the number of neuron network calls.

For this reason we created two test groups. Test set I contained only the basic elements of Hungarian numbers from six speakers. Each uttered the 26 elements twice, giving a total of 312 occurrences, while test set II contained numbers under 100 (169 test cases in all).

5.2 Results

Two things were important in the comparison. First, we had to see how good the method was in scoring correct hits. Second, the number of phoneme-classifying ANN calls made in this task. Actually, the key quantity here for evaluating a method's performance is the lowest number of ANN calls when its performance is maximal.

As the entire hypothesis space is enormous ($> 10^7$ for an average utterance) our goal is to drastically reduce it. The methods tested here require different types of parameters for optimal performance, hence they have to be listed individually.

5.3 Results of using the standard algorithms

The results of each method employed in trials are listed below.

5.3.1 Stack decoding

This method performed surprisingly well on the first test set. Extending the best-scoring of all hypotheses can be regarded as a heuristic, which performs very well with a short utterance, but on longer words it proved unsatisfactory. On the second set (whose elements were much closer to real-life examples) it yielded the worst

	hits (312)	ANN calls on set I	hits (169)	ANN calls on set II
5000	304	1,124,024	141	27,353,614
1000	304	1,124,024	139	10,278,189
500	304	735,135	137	6,798,157
250	303	661,214	136	4,135,990
100	295	562,460	136	2,039,124
50	279	500,748	127	1,148,680
25	260	354,077	124	670,369
10	210	225,152	80	281,704

Table 1: Stack decoding algorithm. The first column indicates the stack size; the best result (the one with the required accuracy and minimum ANN calls) is in bold.

results of all. Overall, this methods works well with short speech utterances but not with long ones. The results can be seen in Table 1.

5.3.2 Multi-stack decoding

The multi-stack decoding method seems most promising. Although it did not perform outstandingly well, it produced fair results and, unlike the other methods mentioned here (with the exception of the flexible A* algorithm) there is significant room for improvement. The main drawback of this method is the fixed stack size. Only in some cases is there a need for a maximum stack size, but here it is applied to all stacks. If we could somehow determine the stack size for each case, the performance of this method would be greatly improved. There results are shown on Table 2.

	hits (312)	ANN calls on set I	hits (169)	ANN calls on set II
100	304	8,808,675	141	7,503,876
50	304	4,421,691	141	3,719,326
25	304	2,173,794	140	1,822,171
20	304	1,732,549	138	1,449,417
15	299	1,292,938	137	1,080,198
10	295	842,595	132	707,777
5	280	416,284	119	348,066
2	240	190,994	90	155,698
1	213	119,576	59	95,938

Table 2: Multi-stack decoding algorithm. Here the parameter shown is the stack size.

5.3.3 Viterbi beam search

Of all the standard algorithms this method worked the best. On the first test set its performance ranked behind that of the stack decoding method, but on the second, more important set it performed very well, producing the lowest run times of the four standard methods. (See Table 3.)

5.4 Results of improvements

Combining standard algorithms

Among the former algorithms only the Viterbi beam and multi-stack decoding methods could be combined (the stack decoding and multi-stack decoding methods are basically different, and the A* algorithm is already an improved version of the stack decoding method). Combining the first two methods led to a more efficient

	hits (312)	ANN calls on set I	hits (169)	ANN calls on set II
25.0	304	2,032,830	141	2,806,010
20.0	304	1,223,316	139	1,394,292
19.0	304	1,098,123	138	1,211,195
18.0	303	983,711	138	1,048,396
17.0	301	884,876	137	912,880
16.0	300	790,772	135	795,547
15.0	297	704,808	134	692,303
10.0	286	380,425	128	341,587
5.0	264	201,408	98	168,941
1.0	229	131,175	71	105,807

Table 3: Viterbi beam search algorithm. Here the parameter shown is the beam width.

algorithm. This idea was included in the other improvements too. Henceforth, when we talk about improving the multi-stack decoding method, we will assume that a Viterbi beam pruning has also been applied.

Phoneme-bound detection

In order to evaluate the probability of a bound we used an ANN, which classified a bound to 80% accuracy. In the first version it achieved its goal. Acting on the first testing set the results approached those of the stack decoding results, and it performed better than the standard algorithms (see Table 4). However, on the

st_{max}	0.50	0.55	0.60	0.65	0.70	0.75	0.80
25	304	304	304	304	304	299	292
	933,993	926,151	918,376	912,275	886,313	788,429	672,395
20	304	304	304	304	304	299	292
	882,358	875,252	868,634	862,734	839,789	752,371	645,656
15	299	299	299	299	299	294	288
	788,599	782,810	777,810	772,591	750,078	684,313	594,605
10	293	293	293	293	293	288	282
	632,134	628,904	626,278	622,681	610,825	566,334	504,831

Table 4: Results using the multi-stack decoding method with the first improvement on test set I.

second set a slighter poorer result was obtained. Surprisingly, this method did slightly worse than the multi-stack decoding method with Viterbi pruning.

In the second version the e^x smoothing technique, however, worked very well.

	Set I	Set II
Stack decoding	735,135	2,039,124
A* heuristic	2,276,965	9,384,119
Multi-stack decoding	1,732,549	707,777
Viterbi beam search	1,098,123	692,303
Multi-stack decoding combined with Viterbi	922,434	474,188
Multi-stack decoding with stack size reduction I	839,789	462,363
Multi-stack decoding with stack size reduction II	749,228	427,212

Table 5: Summary of the best performances of all the methods used

On the first test set it produced almost as good a result as the stack decoding algorithm, and on the second it had the smallest run time. We can say that this novel method is definitely better than the standard algorithms. (Overall, the formula $\min(3 + e^{45.0 \cdot p + 32.3}, 20)$ produced the best results.)

The best results of all methods can be seen on Table 5.

6 Conclusion

In this paper our goal was to study the search problem of speech recognition tasks, compare the standard algorithms and look for ways of improving them. Examining the test results, it is clear that we can indeed marry standard algorithms without loss of accuracy, and with a marked improvement in performance. The novel method presented here proved to be more efficient, and matched or outdid the performance of the others.

Hopefully it could be further refined by using automatic parameter determination or changing the exponential model function to some other. This will be the subject of future work.

References

- [1] L.R. BAHL, P.S. GOPALAKRISHNAN, R.L. MERCER, *Search issues in large vocabulary speech recognition*, Proceedings of the 1993 IEEE Workshop on Automatic Speech Recognition, Snowbird, UT, 1993.
- [2] L.R. BAHL, F. JELINEK AND R. MERCER, *A Maximum Likelihood Approach to Continuous Speech Recognition*, IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 179-190, 1983(2)
- [3] C.M. BISHOP, *Neural Networks for Pattern Recognition*, Clarendon Press, Oxford, 1995.

- [4] P.E. HART, N.J. NILSSON AND B. RAPHAEL, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE Transactions on Systems Science and Cybernetics, pp. 100-107, 1968, 4(2)
- [5] P.E. HART, N.J. NILSSON AND B. RAPHAEL, *Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"*, SIGART Newsletter, No. 37, pp. 28-29, 1972.
- [6] F. JELINEK, *Statistical Methods for Speech Recognition*, The MIT Press, 1997.
- [7] A. KOCSOR, L. TÓTH AND A. KUBA JR., *An Overview of the Oasis Speech Recognition Project*, Proceedings of ICAI '99, pp. 95-102, Eger-Noszvaj, Hungary, 1999.
- [8] L. TÓTH, A. KOCSOR AND K. KOVÁCS, *A Discriminative Segmental Speech Model and its Application to Hungarian Number Recognition*, P. Sojka, I kopecek, K. Pala (eds.): *TSD'2000, LNAI 1902*, pp. 307-313, 2000.

Implementing Global Constraints as Graphs of Elementary Constraints*

Dávid Hanák[†]

Abstract

Global constraints are cardinal concepts of $\text{CLP}(\mathcal{FD})$, a constraint programming language. They are means to find a set of integers that satisfy certain relations. The fact that defining global constraints often requires the knowledge of a specification language makes sharing constraints between scientists and programmers difficult. Nicolas Beldiceanu presented a theory that could solve this problem, because it depicts global constraints as graphs: an abstraction that everyone understands.

The abstract description language defined by the theory may also be interpreted by a computer program. This paper deals with the problematic issues of putting the theory into practice by implementing such a program. It introduces a concrete syntax of the language and presents three programs understanding that syntax. These case studies represent two different approaches of propagation. One of these offers exhausting pruning with poor efficiency, the other, yet unfinished attempt provides a better alternative at the cost of being a lot more complicated.

1 Introduction

Constraint Logic Programming (CLP, also referred to as Constraint Programming, CP) [4] is a family of logic programming languages, where a problem is defined in terms of correlations between unknown values, and a solution is a set of values which satisfy the correlations. In other words, the correlations *constrain* the set of acceptable values, hence the name. A member of this family is $\text{CLP}(\mathcal{FD})$, a constraint language which operates on variables of integer values. Like $\text{CLP}(\mathcal{X})$ solvers in general, $\text{CLP}(\mathcal{FD})$ solvers are embedded either into standalone platforms such as the ILOG OPL Studio [9] or host languages, such as C [3], Java [5], Oz [6] or Prolog [8, 2].

In $\text{CLP}(\mathcal{FD})$, FD stands for *finite domain*, because each variable has a finite set of integer values which it can take. These variables are connected by the constraints,

*The results reported in this paper were presented at the CS² conference held at Szeged, July 1-4, 2002.

[†]Budapest University of Technology and Economics, Dept. of Computer Science and Information Theory, e-mail: dhanak@cs.bme.hu

which propagate the change of the domain of one variable to the domains of others. A constraint can be thought of as a “daemon” which wakes up when the domain of one (or more) of its variables has changed, propagates the change and then falls asleep again. This change can be induced either by an other constraint or by the *distribution* or *labeling* process, which enumerates the solutions by successively substituting every possible value into the variables. Constraints can be divided into two groups: simple and global constraints. The former always operate on a fixed number of arguments (like $X = Y$), while the latter are more generic and can work with a variable number of arguments (e.g., “ X_1, X_2, \dots, X_n are all different”).

Many solvers allow the users to implement user-defined constraints. However, the specification languages vary. In some cases, a specific syntax is defined for this purpose, in others, the host language is used. There are several problems with this. First, CLP(\mathcal{FD}) programmers using different systems could have serious difficulties sharing such constraints because of the lack of a common description language. Second, to define constraints, one usually has to know the solver in greater detail than if merely using the predefined ones. Inspired by these problems, Nicolas Beldiceanu suggested a new method for defining and describing global finite domain constraints [1]. After studying his theory, I decided to put it into practice by implementing a parser of Beldiceanu’s abstract description language (ADL), as an extension to the CLP(\mathcal{FD}) library of SICStus Prolog [7, Section CLPFD], a full implementation of the CLP(\mathcal{FD}) language.

The paper is structured as follows. Section 2 introduces the theory of Beldiceanu, explains how constraints may be represented by graphs and describes the ADL in some detail. Section 3 specifies the concrete syntax of the language used by the implementation, Section 4 presents the implemented programs capable of understanding such a description. Section 5 gives some ideas about the possible directions of future research and development, and finally Section 6 concludes the paper.

2 The Theory

In [1], Beldiceanu specifies a description language which enables mathematicians, computer scientists and programmers of different CLP systems to share information on global constraints in a way that all of them understand. It also helps to classify global constraints, and as a most important feature, it enables us to write programs which, given only this abstract description, can automatically generate parsers, type checkers and propagators (pruners) for specific global constraints.

Beldiceanu has also defined a large number of constraints in the ADL. Most of them are already known, but the slight modification of existing descriptions has resulted in several new constraints. The potential of these modifications arose only with the use of this schema.

Section 2.1 introduces the essential concepts of Beldiceanu’s theory, Section 2.2 presents the most important features of the ADL, finally Section 2.3 illustrates the usage through the simple example of the widely used `element` constraint.

2.1 Representing Constraints as Graphs

In order to create an inter-paradigm platform, Beldiceanu reached for a device that is abstract enough and capable of depicting relations between members of a set: the *directed graph*. Before we can show how graphs can represent global constraints, three concepts have to be introduced:

1. The *initial graph* is a regularly structured graph, which is characteristic of the constraint and the number of arguments¹, but is independent from the specific values of the arguments.
2. The *elementary constraint* is a very simple constraint with few arguments, such as $X = Y$.
3. The *graph properties* are restrictions on the number of arcs, sources, connected components, etc.

The description of a constraint specifies how the *initial graph* should be built. Its vertices are assigned one or more variables from the constraint, while the arcs connecting the vertices are generated according to a regular pattern. Finally the chosen elementary constraint is assigned to each arc. The variable belonging to the start point of the arc will become the first argument of the elementary constraint, while the variable assigned to the endpoint will become the second argument. Note that in general, the elementary constraint need not be binary, if it has more arguments, then a hypergraph is built using arcs with the required number of endpoints.

Every distinct instantiation of the constraint arguments results in a separate instance of the constraint. For every such instance, a different *final graph* is derived from the common initial graph by keeping those arcs for which the elementary constraint holds. If a vertex is left without connecting arcs, the vertex itself is also removed. The global constraint succeeds if and only if the specified graph properties hold for this final graph.

The graph of a simplified variant of the element constraint can be seen in Figure 1. This constraint serves as an example throughout this paper, and it is explained in detail in Section 2.3. For now, it is enough to know that it succeeds if its first argument, a single variable (denoted by A in the figure), is equal to a member of its second argument, a list of values (denoted by B, C, D and E). The required graph property is that the number of arcs should be exactly one.

2.2 The Abstract Description Language (ADL)

The most important feature of the ADL is the ability to describe how the initial graph has to be generated, what is the elementary constraint to be assigned to the arcs, and what graph properties must hold for the final graph.

Beside these, the ADL gives means to limit the set of values to be accepted in the constraint arguments, too. We have to specify the type of each argument, and

¹As already mentioned, global constraints may (and usually do) have variable number of arguments.

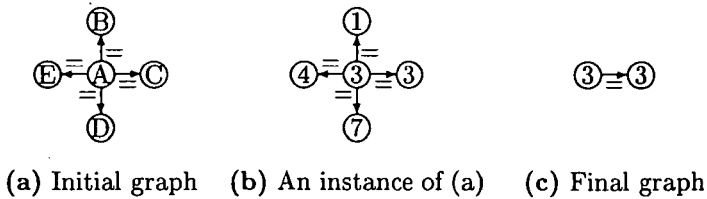


Figure 1: The graph of the simplified element constraint

we may also pose further restrictions on the values. Any concrete application of the constraint that violates these preconditions will be considered as erroneous.

The syntactic order of these language elements in a concrete definition reflects the order in which they are interpreted: the type and value restrictions are followed by the graph generation parameters, finally the required graph properties are specified. The following paragraphs discuss the language features in the very same order.

2.2.1 Argument type restrictions.

According to the schema of Beldiceanu, all arguments of the global constraint must be typed. There are three simple data types and a compound type, which are widely used. An argument of type

int is a constant integer;

atom is a character sequence (just like a Prolog atom);

dvar is a domain variable (which could also be a constant as a special case);

collection(Attr₁-Type₁, Attr₂-Type₂, ...) is an *ordered list of items*, each item being a *set of labeled attributes*, where the attribute associated with the label Attr_{*i*} (if any) has type Type_{*i*}, for each *i*. This type specification does not require the items of such a collection to have all the attributes specified and also allows them to have additional attributes. It only requires the values of the given attributes to have the right type. An example collection and its type specification (taken from [1]) is shown in Figure 2.

There are other compound data types, too, like **list** or **term**, which are rarely used in the numerous existing constraint descriptions.

2.2.2 Argument value restrictions.

In addition to type restrictions it is also possible to specify preconditions on the *values* of the arguments. These conditions can be expressed with the following formulæ:

Name Relop Expression means Name must be in relation Relop with Expression. Here Name is the name of either an argument or an attribute of a collection, in the form Coll.Attr, Coll being the collection. Relop is a relational

The type **RECTANGLES** corresponds to a collection of rectangles, each rectangle being defined in both dimensions by its origin and either its size or its end. The following is the type definition of **RECTANGLES** and a sample instance of it, that contains two rectangles, one given with its size, the other with its endpoint. (The attributes of each rectangle are separated by spaces, the two rectangles are separated by a comma.)

```
RECTANGLES: collection(ori1-dvar, siz1-dvar, end1-dvar,
                       ori2-dvar, siz2-dvar, end2-dvar)
```

```
RECTANGLES = { ori1-5   siz1-20   ori2-5   siz2-10,
                ori1-25  end1-45   ori2-15  end2-25 }
```

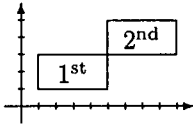


Figure 2: An example collection

operator, like \neq or \geq . *Expression* is an arbitrary expression consisting of constants, other names and mathematical operators.

Name in $\{List\}$ means *Name* (same as before) must appear in *List*, a list of comma separated constants.

$\text{distinct}(Coll/Attr)$ means that for any two items in the collection *Coll* the values of attribute *Attr* must be different.

$\text{required}(Coll.Attr)$ means that all items in collection *Coll* must have attribute *Attr* specified.

There are several other value restricting statements, but those are seldom used.

2.2.3 Graph generation parameters.

The initial graph generation consists of three phases. In the first phase the vertices are created, in the second phase they are connected by arcs, and in the third phase, the specified elementary constraint is assigned to each arc.

In the most common case, one has to specify a single *input collection* to create the vertices: to each element of this collection a vertex is assigned. The collection may either be a constraint argument or it can be built for this purpose. The vertices thus created provide the input of the *arc generator*, which manages the second phase. Each generator incorporates a regular pattern, which is reflected in

the created set of arcs. The arity of the arcs is characteristic of the generator, and it must also match the arity of the specified elementary constraint.

In general, the arc generator may require the vertices to be divided into disjoint subsets. In that case, not one but several input collections must be specified, each of these is mapped to a separate subset of vertices. Currently, most existing arc generators need a single set of vertices (i.e., one collection) as an input, and there are two of them expecting two.

Figure 3 shows four example arc generators. All of these generators create binary arcs, which means that they can be used with binary elementary constraints only. (This is the most common case.) The loop generator connects each vertex to itself. The path generator exploits that the collections are ordered lists, and connects the first vertex to the second, the second to the third, and so on. The clique generator connects all vertices to all others by default, but it can have a relational operator as an argument, in which case it only connects vertices with indices which sustain the relation. Such a case is shown in the figure. The product generator gets two sets as an input, and connects all the vertices in the first set to all the vertices in the second set. This generator can also have a relational operator as an argument.

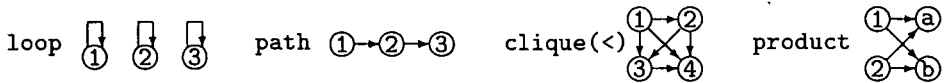


Figure 3: Arc generators

The elementary constraint, the third ingredient of the graph generation, is basically a mathematical relation containing symbolic references to values assigned to the vertices (i.e., the endpoints of the arcs).

A constraint definition contains three terms to specify the graph to be generated. We have to determine the input collection(s), select the arc generator by its name, and define the elementary constraint assigned to the arcs.

2.2.4 Graph property requirements.

These statements also have the form of an equation, with a graph property name on the left hand side, constants and arguments of the global constraint on the right. Let us see several graph properties:

- nvertex** is the number of vertices;²
- narc** is the number of arcs;
- ncc** is the number of connected components;
- nscc** is the number of *strongly* connected components;

²This property is sensible to examine, because unconnected vertices are removed from the graph, therefore it is not necessarily equal to the size of the input collection.

nsource is the number of sources (those vertices which do not have arcs leading into them);

nsink is the number of sinks.

2.3 An Example – The element Constraint

The **element** constraint is one of the most common global constraints. It receives a single item and a set of items as arguments and it succeeds iff the item is a member of the set. In some implementations, both the item and the elements of the set may be domain variables, but in the following interpretation the elements of the set must be constants. The formal definition of **element** according to [1] is shown in Figure 4, an instance of its graph with specific arguments was presented in Figure 1. It can be explained as follows.

1. The **element** constraint has two arguments (line 1).
2. The first, called **ITEM** is a collection with two attributes, **index** and **value**, both are domain variables (line 2). The second argument, called **TABLE** is also a collection with two attributes, also called **index** and **value**, but these are constants (line 3).
3. The following restrictions must hold:
 - both attributes of both collections must be specified in all items (lines 4–5);
 - there must be exactly one item in the **ITEM** collection (line 6);
 - the indices in both collections must be between 1 and the size of **TABLE** (lines 7–8);
 - all indices in **TABLE** must be distinct (line 9).
4. The arc generator is **product** (line 11), which requires two collections as its input, namely **ITEM** and **TABLE** (line 10).
5. The elementary constraint assigned to the arcs appears in lines 12–13. It is to be read like this: the value assigned to the first endpoint of the arc ([1]) is a member of the **ITEM** collection, and its attributes labeled as **index** and **value** must both be equal to the equivalent attributes of the value assigned to the second endpoint ([2]), which is a member of the **TABLE** collection. The syntax looks a bit weird and perhaps even confusing. We will further discuss this question in Section 3.1.
6. The number of arcs must be exactly 1 in the final graph (line 14).

```

1 Constraint:      element(ITEM, TABLE)

2 Arguments:      ITEM:  collection(index-dvar, value-dvar)
3                 TABLE: collection(index-int, value-int)

4 Restrictions:   required([ITEM.index, ITEM.value]),
5                 required([TABLE.index, TABLE.value]),
6                 |ITEM| = 1,
7                 ITEM.index ≥ 1, ITEM.index ≤ |TABLE|,
8                 TABLE.index ≥ 1, TABLE.index ≤ |TABLE|,
9                 distinct(TABLE/index)

10 Arc input:     ITEM, TABLE
11 Arc generator:  product
12 Arc constraint: ITEM.index[1] = TABLE.index[2] ∧
13                 ITEM.value[1] = TABLE.value[2]

14 Graph property: narc = 1

```

Figure 4: The element constraint in abstract syntax

Note. It might seem strange to define ITEM as a collection when it must have exactly one element (line 6). However, passing the index and value as two separate arguments of the constraint would be less symmetric with respect to TABLE. Another advantage is that ITEM, being a collection, can serve directly as an input for the product arc generator.

3 The Concrete Syntax

In order to be able to put the theory into practice, we had to define a concrete syntax of the language. The chosen representation closely resembles the abstract syntax, but follows the syntax of Prolog, too. This has the advantage that it can be effortlessly parsed by a Prolog program.

This work has helped to discover some weaknesses of the ADL. First, it turned out that the semantics of the *distinct* operator is unclear in certain contexts, because it is under-specified. Second, as it was already noted at the end of the previous section, the syntax of the elementary constraint specification can be confusing.

Section 3.1 covers the two problematic issues and suggests a solution to both. Section 3.2 discusses the concrete syntax itself, illustrated by the updated version of the already familiar *element* example.

3.1 Clarifying the Language Specification

3.1.1 The problem of the distinct operator.

Let us consider the following type declaration of a collection of collections:

```
COLL: collection(c-collection(val-int))
```

Such a data structure can be used to model different data semantics. Two of these are the following:

1. The inner collections depict sets, thus each of them must have distinct elements, but the same element can appear in more than one collection.
2. The inner collections represent partitions, i.e., pairwise disjoint subsets of a single superset. In this case all elements of all the inner collections must be pairwise different.

Since the used data structure is the same in both cases, the distinction must be made using value restrictions, more specifically *distinct* statements. Unfortunately, it is clear that we cannot express both with the *distinct(COLL/c/val)* statement. Moreover, it is unclear which of the two semantics the statement expresses. The inability to precisely determine the value sets *distinct* operates upon leads us to the definition of two concepts in the following paragraph.

3.1.2 Selectors and designators.

When we refer to attributes of items of collections, sometimes we want to reach single values, in other cases we need the list of values of all items within the collection. The *required* and *distinct* operators are good examples of the two possibilities, respectively.

Keeping the notation of [1], which uses the term *designator* to refer to a sequence of names selected by slashes, let us introduce two new concepts, defined with BNF notation as follows:

```
selector ::= Coll | selector . Attr
```

```
designator ::= selector | designator / Attr
```

Selectors can be used to point out single values. They can be used to state something about values of items of a collection *separately*. Designators, on the other hand, point to a list of values. They can be used to state something about the values in all the items of a collection *together*.

By starting a designator with a selector, we express that we want to divide the list of all the values into sublists and state something about these sublists separately. The division points are determined by the selector part of the designator. To clarify this, Figure 5 shows a somewhat degenerated collection as a tree, along with two designators and the corresponding sublists marked with ovals.

C: collection(a-collection(b-collection(c-int)))

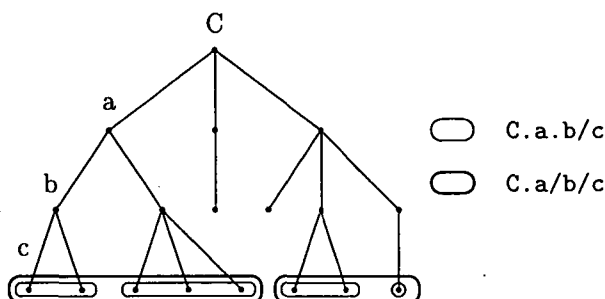


Figure 5: The meaning of designators

In certain contexts only selectors are accepted. Among others, such places are where the dot notation was already used, such as the argument of `required`, or `TABLE.index ≥ 1` in Figure 4. In the latter we want to express that all values labeled as `index` must be greater than or equal to 1, separately.

Elsewhere designators are required. The argument of `distinct` is such a place. One would also use a designator to count those items of a collection which possess a certain attribute. Then one needs to write `|COLL/attr|`, because `COLL/attr` brings all the items with `attr` attribute together into a list, and `|...|` returns the length of this.

Now let us return to the problem of `distinct`. In the example presented there, the statement `distinct(COLL.c/val)` means that for all `c` collections separately, the `val` values must be distinct, but the same value can appear in more than one collection (case 1). `distinct(COLL/c/val)`, on the other hand, means that the *list of all values* in all collections must be distinct (case 2).

3.1.3 The elementary constraint notation.

As we have seen in Figure 4, `ITEM.value[1]` means the value of the `value` attribute of the first argument which is the `ITEM` collection. Thus we can say that the general form is something like `Coll.Attr[ArgIndex]`. This is rather confusing and does not resemble any of the notations we are used to:

- the specification of `ITEM` is redundant, because it is well known from the graph structure that the first argument of the elementary constraint *must be* an item of that collection;
- the position of the index 1 between the brackets is misleading because this notation suggests some kind of array indexing, which is not the case.

We would be better off with a notation like `Args[1].value` (where `Args` would be an array of all arguments of the elementary constraint) or `Arg1.value`. As we will see, the concrete syntax uses a very similar notation.

3.2 The Prolog-like Concrete Syntax

As stated in the introduction of Section 3, the chosen representation, while closely resembling the abstract syntax, follows the syntax of Prolog. Hence, each global constraint is described by a Prolog clause with seven arguments, as shown in Figure 6 for the `element` constraint. These arguments are the following:

1. the name and arguments of the global constraint (as a Prolog term);
2. the list of type restrictions of the form *Arg-Type* where *Arg* is the name of the argument and *Type* is the type specification;
3. the list of value restrictions in a form very similar to the abstract syntax, with the exception of `|COLL|` which should be written as `size(COLL)`, and all the relational operators must be written in Prolog notation;
4. the arc generator input (a list of collections);
5. the name of the arc generator;
6. the elementary constraint in the form *Args => Body*, where *Args* is a collection of the arguments of the elementary constraint and *Body* is the constraint itself (`#=` and `#/\` are operators of the host language, basically they mean `=` and `^`, respectively);
7. the list of graph properties.

Inline collections have a somewhat different syntax than the one in Figure 2. They can be written as follows (note the difference in the use of commas and semicolons):

- a collection has the form `{ Item1 ; Item2 ; ... };`
- each *Item_i* above has the form *Attr₁-Val₁ , Attr₂-Val₂ , ...* where *Attr_i* is an attribute name and *Val_i* is a value.

The lines of Figure 6 correspond respectively to the lines of Figure 4, and the definition as a whole should be self-explanatory. However, two things are worth mentioning.

One is that we have chosen to represent the arguments of the elementary constraint as items of a collection. In the body, we need to refer to these items and their attributes. The collection can be broken up into separate items simply by writing a pattern. But to access the attributes of these items, we must call for a trick: by wrapping *A* in braces, we create a collection with a single item, therefore `{A}.index` will expand to the index of this single element.

```

1 graphfd:global(element(Item, Table),
2     [ Item-collection(index-dvar, value-dvar),
3       Table-collection(index-int, value-int)           ],
4     [ required(Item.index), required(Item.value),
5       required(Table.index), required(Table.value),
6       size(Item) == 1,
7       Item.index >= 1, Item.index <= size(Table),
8       Table.index >= 1, Table.index <= size(Table),
9       distinct(Table/index)                             ],
10    [ Item, Table                                         ],
11    product,
12    {A;B} => {A}.index != {B}.index #/\
13            {A}.value != {B}.value,
14    narc = 1).
```

Figure 6: The element constraint in concrete syntax

The other point to note that the relational operator `!=` comes from the SICStus CLP(\mathcal{FD}) library, therefore, after expanding the selectors, the statement will become a valid CLP(\mathcal{FD}) expression. The advantages of this will be discussed in Section 4.2.

4 The Implementation

The schema created by Beldiceanu allows us to test whether the relation expressed by a global constraint holds for a given set of concrete arguments. However, it does not deal with the more important case where only the domains of the arguments are specified, but their specific values are unknown. In such a case we need an algorithm to prune the domains of the arguments by deleting those values that would certainly result in a final graph not satisfying the properties. This question is fundamental in practical applications, therefore it is addressed by this section.

Development was launched with two goals in mind. The first task was to implement a relation checker, a realization of the testing feature offered by the schema, and a dumb propagator built on this checker. By and large, this task is finished, the results are presented by Section 4.1.

The second task was to implement a direct propagator capable of pruning variable domains based on an analysis of the current state of the graph, with the

required properties in view. This task is much bigger, the development is still in an early stage. Its current state and features are introduced by Section 4.2.

Both tasks are implemented in SICStus Prolog [7], extending its $\text{CLP}(\mathcal{FD})$ library by utilizing the interface for defining global constraints. This allows thorough testing of both the program and the theory itself in a trusted environment.

4.1 The Complex Relation Checker and the Generate-and-Test Propagator

The first stage was to implement the complex relation checker, a program that checks whether the relation defined by the global constraint holds for a given set of values, but does no pruning at all. It includes the following features:

- complete type checking (dvar is interpreted as int);
- full support of selectors and designators introduced in Section 3.1;
- support for value restriction with the most frequent statements:
 - distinct and required; plus
 - arbitrary Prolog calls which must succeed for the restriction to hold;
 - size(...) is replaced by the length of a collection or list.
- full set of built-in arc generators;
- extensive set of supported graph properties.

When called, the relation checker is given a constraint with fully specified arguments, and reports the result of the type check, the restriction check, and whether the graph properties hold for the final graph. The output of two example runs can be seen in Figure 7. In the first case, the first argument appears in the collection passed in the second argument, while in the second case it does not.

The checker was used to test the formal description of several constraints, whether they really conform to their expected meaning, and some errors in their specification have already been discovered (these will not be discussed here).

The second stage was to amend the relation checker with a generate-and-test propagator. The idea is that whenever the domain of a variable changes, all possible value combinations of the affected constraint's arguments are tested with the relation checker, and only the values that passed the test are preserved. This classical but extremely inefficient method for finding solutions gives us full and exhaustive pruning.

The usage and output of the generate-and-test propagator is the same as that of the direct propagator, which is introduced in the next section (see Figure 8).

```

Testing element({index-2,value-3}, {index-1,value-1 ; index-2,value-3}).
Type checking passed.
Type restrictions held.
Graph properties held.
Relation is sustained.

Testing element({index-2,value-1}, {index-1,value-1 ; index-2,value-3}).
Type checking passed.
Type restrictions held.
Graph properties failed.
Relation is not sustained.

```

Figure 7: Output of the complex relation checker

4.2 The Direct Propagator

Generate-and-test propagation is naturally out of the question in any practical applications. The direct propagator is the first step towards an efficient, applicable pruner. Here the line of thought is reversed: we assume that the constraint holds, and from the required graph properties we try to deduce conclusions regarding the domains of its variables.

4.2.1 Propagation in theory.

The question that naturally arises is the following: how the changes of domains can be propagated given a graph representation of the constraint. As mentioned in the Introduction, constraints behave like daemons which wake up when the domain of the affected variables change. The propagator – using the programming interface of $\text{CLP}(\mathcal{FD})$ – can be set up so that it is notified whenever a constraint wakes up. On these occasions it must check the graph corresponding to the constraint and classify its arcs into three groups:

1. arcs with the assigned elementary constraint being *known to hold* – i.e., they will appear in the final graph;
2. arcs with the assigned elementary constraint being *known to fail* – i.e., they will be left out of the final graph;
3. arcs with the assigned elementary constraint being yet uncertain.

This classification can be completed gradually because the $\text{CLP}(\mathcal{FD})$ system is monotonic, which means that values can only be removed from a domain. As a result, a value is removed only if it is definitely not a solution, because it cannot be re-added later.

To propagate the constraint, we have to look at this semi-determined graph and the required graph property together, and try to tell something about the still uncertain arcs. This process is called the *tightening* of the graph. In order to ensure that the graph properties hold, some of the uncertain arcs must be removed from the final graph, others must be made part of it. This causes the corresponding elementary constraints to be forced into success or failure, thus pruning the domains of the variables. The global constraint finally becomes entailed when there are no uncertain arcs left.

Because of the character of this propagation algorithm, elementary constraints are chosen to be *reifiable* CLP(\mathcal{FD}) constraints, as shown in Figure 6. Reifiable constraints are connected with a Boole variable, and succeed *if and only if* the Boole variable has a value of 1. This use of reifiable constraints has several advantages. For one, a wide range of predefined constraints is available, already at this early stage of development. For another, the algorithm must be able to determine whether an elementary constraint holds or fails, or force it into success or failure, and the Boole variable linked to the reifiable constraints serves exactly that purpose.

To figure out how to tighten the graph at each step, that is, to find the rules of pruning, we need to study each graph property separately. There are simpler properties, such as prescribing the number of arcs, for which finding these rules is not very problematic (see below). A few of these are already handled by the propagator. The more complex properties, like constraining the difference in the vertex number of the biggest and smallest strongly connected components, the pruning rules for these are a lot more complicated.

4.2.2 Propagation of the $\text{narc} = N$ property.

Let us assume that the required graph property is $\text{narc} = N$, where N is a positive integer. Let S be the set of arcs which are known to be part of the final graph, and let U denote the set of the still uncertain arcs. Then we have to take the following action:

- if $|S| > N$, fail, because there are already too many arcs;
- if $|S| = N$, force every arc in U to failure;
- if $|S| + |U| < N$, fail, because N cannot be reached any more;
- if $|S| + |U| = N$, force every arc in U to success;
- otherwise do nothing.

4.2.3 Example run.

Running the direct propagator on the `element` constraint is possible because it uses exactly this graph property. An example run can be seen in Figure 8. The first call determines that the A-B element must appear in the list, and we get that A must be between 1 and 3, while B must be either 2, 6 or 9. The second call we also ask

the CLP(\mathcal{FD}) environment to enumerate all solutions by labeling the variable A, and, as we could expect, we get the three correct solutions.

```
| ?- graph_global(element({index-A,value-B},
    {index-1,value-6 ; index-2,value-2 ; index-3,value-2})).
A in 1..3, B in{2}\/{6} ? ;
no

| ?- graph_global(element({index-A,value-B},
    {index-1,value-6 ; index-2,value-2 ; index-3,value-2})),
    labeling([], [A]).
A = 1, B = 6 ? ;
A = 2, B = 2 ? ;
A = 3, B = 2 ? ;
no
```

Figure 8: Running the direct propagator

The current implementation can handle four graph properties, these are `narc`, `nvertex`, `nsource` and `nsink`. Fortunately, a large number of descriptions relies only on the first two, thus many different constraints can already be propagated. Without going into details, such constraints are among, `disjoint`, `common`, `sliding_sum`, `change`, `smooth`, `inverse`, and variants of these.

Current work is concentrated on the perfection of the propagation of these properties, and on the study of the `nscc` (number of strongly connected components) and related properties, which are also heavily used in the existing descriptions. The rest of the properties are only required by a minority of the constraints.

This propagator, although still not efficient enough to be useful in practical applications, may serve as a prototype for more effective implementations. A few thoughts on this issue are shared in the next section.

5 Future Work

Pruning rules for more of the graph properties are to be worked out. The existing rules also need to be improved in certain cases. This will be the objective of an international project hopefully starting in Autumn 2003.

Using reifiable constraints as elementary constraints poses a problem: they do not necessarily provide a pruning as strong as expectable. Such a case can be seen on Figure 9. What we see here, is that 1 is not excluded from the domain of A, although it could be. The problem is that forcing the and-ed elementary constraint of `element` (Figure 6, lines 12–13) into failure is not enough to do that. We would get better pruning if we could write something like:

$\text{arc exists} \iff \text{indices are equal}$
 $\text{arc exists} \implies \text{values are equal}$

But implication does not conform with the concept of elementary constraints. This problem requires further study.

```
| ?- graph_global(element({index-A,value-B},
    {index-1,value-6 ; index-2,value-2 ; index-3,value-2})),
    B #= 2.
B = 2,
A in 1..3 ?
```

Figure 9: Weak propagation of reifiable constraints

Efficiency matters need to be considered more carefully when implementing further propagators. One way to increase efficiency, as suggested by Beldiceanu, could be to abandon the thought of a common propagator, that is able to parse such descriptions and prune in run time, and implement a *pruner algorithm generator* instead. This generator would take the description and convert it into a piece of code that does the pruning. This would shift the execution of complicated graph algorithms into compile time, where efficiency is a smaller issue. How this can be done must be worked out yet.

6 Conclusions

The paper began with the introduction of a theory first described in [1] that enables us to represent global constraints as regular graphs of the same elementary constraint. It was shown how the definition of a global constraint looks like, what restrictions and requirements may appear in it, and how the representative graph is built by it.

Then the concrete syntax of the language developed for the implementation was presented. First, attention was drawn to two problems with the ADL specification, and solutions to them were suggested, too. Second, the concrete syntax itself was illustrated.

The last part of the paper described the results on constraint checking and propagation. The first of these was the complex relation checker capable of testing whether a constraint holds for a given set of values. The second, based on this, was the generate-and-test propagator, which implements exhaustive propagation for a large number of graph properties, but with very low efficiency. The result of the third, more interesting approach was the direct propagator, which was considered as a step towards efficient algorithms of constraint pruning. This deals with semi-determined constraint graphs and the enforcement of uncertain arcs in order to satisfy the required graph properties.

Acknowledgements

I would like to thank Nicolas Beldiceanu for his theory and his ideas on the propagation of graph properties. Péter Szeredi, my supervisor always directed my attempts at research and writing with patience yet with great tenacity. Thanks are also due to the anonymous reviewer whose remarks led to major improvements in the paper.

References

- [1] Nicolas Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. In *Principles and Practice of Constraint Programming*, pages 52–66, 2000.
- [2] Daniel Diaz and Philippe Codognet. A minimal extension of the WAM for clp(FD). In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, 1993. The MIT Press.
- [3] ILOG. *ILOG Solver 5.1 User's Manual*. ILOG s.a. <http://www.ilog.com>, 2001.
- [4] Joxan Jaffar and Spiro Michaylov. Methodology and implementation of a CLP system. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the 4th International Conference*, pages 196–218, Melbourne, May 1987. MIT Press. Revised version of Monash University technical report number 86/75, November 1986.
- [5] Vincenzo Loia and Michel Quaggetto. Embed finite domain constraint programming into Java and some Web-based applications. *Software—Practice and Experience*, 29(4):311–339, April 1999.
- [6] Gert Smolka. Constraints in OZ. *ACM Computing Surveys*, 28(4es):75, December 1996.
- [7] Swedish Institute of Computer Science, Uppsala, Sweden. *SICStus Prolog User's Manual*, 2003. <http://www.sics.se/isl/sicstuswww/site/documentation.html>.
- [8] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1989.
- [9] Pascal van Hentenryck, Laurent Michela, Laurent Perron, and Jean-Charles Régin. Constraint programming in OPL. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 98–116, September 29 - October 1 1999.

On Implementing Relational Databases on DNA Strands

István Katsányi*

Abstract

This work describes the theoretical bases of the implementation of relational databases in test tubes, using an abstract model of molecular computing. It specifies the representation of relations and the execution program of the relational algebra (RA) operations. We investigate the possibilities of practical usage of the proposed model as well as the bounds of it.

Key words: Molecular computing, theory of computing, relational database.

1 Introduction

In the last decade molecular biology has become the fastest growing discipline in the world. Some of the results are widely known, let us only mention the major breakthroughs in the Human Genome Project and nanotechnology. The progress made possible the birth of a new branch of science, that is called molecular computing (or DNA computing). Leonard M. Adleman published a paper [1] in 1994, which later became the foundation-stone of this new subject. In his article Adleman demonstrates how can a classic NP-complete problem: the problem of searching for a Hamiltonian path in a directed graph can be solved in polynomial time using the techniques of molecular biology and DNA strands. He outlines the great opportunity laying in the large computing power and the extremely compact data storage. In a test tube there can be performed as much as 10^{16} operations in a second. That is much more than current supercomputers can execute. In a litre of water DNA strands can encode 10^8 terabytes, and we can perform associative searches on the data in constant time.

In the past years many papers dealing with the computing power of DNA were published. However, only a few articles studied the possibility of data storage and processing (see e.g. [2], [3]). Recently two papers ([4], [5]) described methods that yielded in an operation that closely resembles to the join operation of relational algebra. In spite of this no one has extensively studied the potentialities of the

*Eötvös Loránd University, Department of General Computer Science, 1117 Budapest, Pázmány Péter sétány 1/C, e-mail: kacsai@ludens.elte.hu

usage of molecular computing in the field of RA. By this work we would like to show the (at least theoretical) possibility of the use of the results of molecular computing in this area. In the *Preliminaries* section we introduce John Reif's RDNA model of biomolecular computing. We define a possible representation of relations using that model in the *Representation* section, and show how to implement the RA operations in this model in the *Operations* section. We close the paper by conclusions and references.

2 Preliminaries

As common in formal language theory, we denote the free monoid generated by a finite set X by X^* . We call X as *alphabet*, elements of X as *letters*, and elements of X^* as *words*. The symbol ϵ means the empty word. The length of a word $u \in X^*$ will be denoted by $|u|$. The cardinality of a set S will also be denoted by $|S|$.

We describe briefly the *RDNA* model introduced by J. H. Reif in [4]. For motivations, connections to molecular biology and complete definitions please refer to the original article. The operations of the model are abstractions of the well understood recombinant DNA operations and basic molecular biology operations. The structural properties of DNA are represented in a structure called *complex*.

We use an alphabet consisting $n \geq 1$ pairs of letters that said to be complementary: $\Sigma = \{a_1, a_2, \dots, a_n, a'_1, a'_2, \dots, a'_n\}$, where a_i and a'_i form a complementary pair for all $i \in [1, n]$. By a *linear string* we mean a word from Σ^* , and by a *loop string* we mean any possible circular rotation of a word from Σ^* . The set of circular rotations of word $u \in \Sigma^*$ is $\{u_2u_1 \in \Sigma^* \mid u_1u_2 = u\}$. A loop string can be represented by any particular instance of the rotated words. For a linear string u define $G(u)$ as a directed graph of $|u|$ vertices and $|u| - 1$ edges, such that the graph consists of one (nonrepeating) directed path, where the consecutive edges are labelled by the consecutive letters of u , from first to last. For a loop string u define $G(u)$ as a directed graph of $|u|$ vertices and $|u|$ edges, such that the graph consists of one directed loop, where the consecutive edges are labelled by the consecutive letters of u . For a set S of linear or loop strings over Σ define $G(S)$ as the union of the disjoint directed graphs $G(u)$ for each $u \in S$. Hence $G(S)$ consists of $|S|$ disjoint directed paths or loops.

Define a *labelled pairing* of the edges of $G(S)$ to be a set μ of unordered pairs of distinct directed edges of $G(S)$ such that (1) μ pairs the starting and ending vertices of the edges as well as the edges itself, (2) no edge appears more than once in μ , and (3) each pair of edges in μ have complementary labels and point to the opposite direction. We define a *complex over S* to be the pair (S, μ) , where μ is a labelled pairing of $G(S)$.

The complex (S, μ) has a naturally defined graph $G(S, \mu)$ derived from the graph $G(S)$ by merging together the vertices i, i' as well as the vertices j, j' for all labelled pairs $((i, j), (j', i'))$ in μ , so the resulting graph has edges in both directions between the two merged nodes. Note that three nodes may be merged into one, for example j and j' would be merged with j'' if the pair $((j, k), (k', j))$ is in μ in addition to

the pair $((i, j), (j', i'))$.

The complex (S, μ) is a *linear complex* if $\mu = \emptyset$ and S is a set of linear strings. The complex (S, \emptyset) is a *single linear complex* if S contains one linear string only, so the graph $G(S)$ is a single directed path.

A complex may be used to model both the information content and also the three-dimensional structure of single- or double-stranded DNA, including hybridization and secondary structure. The use of complexes allows modelling the effect of various recombinant DNA operations, and thus providing rigorous definitions of recombinant DNA operations.

Operations of the RDNA model

We use a slightly different notation for the operations than Reif uses. We also extend the list of operations by the operations *Prepare*, *Assign* and *Amplify*. By a test tube we mean a multiset of *connected* complexes, where a complex (S, μ) is said to be connected, if its graph $G(S, \mu)$ is connected. We call two sets of complexes equivalent, if the union of the graphs of the complexes in each set are isomorphic. The allowed operations (also called instructions) are the following:

1. *Prepare*. The operation $T := \text{Prepare}(S)$ prepares the test tube T containing the linear complex (S, \emptyset) from the set of linear strings $S \subset \Sigma^*$.
2. *Assign*. We use the usual $:=$ operator for assigning values for test tubes.
3. *Merge*. After the operation $\text{Merge}(T_1, T_2)$ the test tube T_1 becomes the union of multisets T_1 and T_2 .
4. *Copy*. $T' := \text{Copy}(T)$ produces a copy of the test tube T containing only linear complexes.
5. *Amplify*. By using $\text{Amplify}(T, n)$ each complex of the multiset T is replaced by at least n identical copies of it, hence the volume of T is multiplied by at least n .
6. *Detect*. $\text{Detect}(T)$ returns true if T is not the empty multiset, false otherwise.
7. *Select*. Operations $T' := \text{Select}_=(T, n)$ and $T' := \text{Select}_\neq(T, n)$ separate the contents of T by the size of the complexes. The size of a complex is the number of nodes in its graph. T' will contain those complexes of T , whose size are equal, (or in case of Select_\neq not equal) to n .
8. *Separate*. Operations $T' := \text{Separate}_{incl}(T, u)$ and $T' := \text{Separate}_{excl}(T, u)$ separate the contents of T by the content of the complexes in it, where u is a word over Σ . The operation may only be applied on a test tube of linear complexes. T' will contain those complexes (S, \emptyset) of T , where there exists (or in case of Separate_{excl} does not exist) a word $v \in S$ such that u is a subword of v .
9. *Cleave*. The operations $\text{Cleave}_{before}(T, \sigma)$ and $\text{Cleave}_{after}(T, \sigma)$ cuts every path of the complexes of T before (resp. after) the edges labelled with $\sigma \in \Sigma$.

If the created complex is not connected, it is replaced by connected complexes equivalent to it.

10. *Anneal*. The *Anneal*(T) operation changes the test tube T nondeterministically. Each complex (S, μ) of T is replaced by a complex of form (S, μ') , where μ' is a superset of μ , and there is no μ'' labelled pairing of $G(S)$ such that $\mu'' \supsetneq \mu'$.
11. *Ligate*. By the use of the operation *Ligate*(T) every complex (S, μ) in T is replaced by a complex (S', μ') , which has the same graph as (S, μ) , except that all vertices that are paired to the same node in μ are merged. That means that any two paths in $G(S)$ that have ending and beginning nodes that are paired to the same (third) node are concatenated.
12. *Denature*. The *Denature*(T) operation replaces every complex (S, μ) in T by the set of connected complexes $\{(\{\alpha\}, \emptyset) \mid \alpha \in S\}$.

3 Representation

In this section we show a method, by which arbitrary finite relations can be represented in test tubes using the devices introduced in the previous section.

Each relation is represented by a unique test tube containing complexes over a common alphabet. The tubes contain mainly linear complexes. During performing certain operations there may occur other structures as well, but these are eliminated by the end of the operation.

Suppose that we have m not necessarily different sets A_1, \dots, A_m , by which the bases of n relations R_1, \dots, R_n are defined. Each relation R_i consists of k_i components: $R_i \subseteq A_{f_{i,1}} \times \dots \times A_{f_{i,k_i}}$ ($i = 1, \dots, n, k_i \geq 1, f_{i,j} \in [1, m]$ for all $j \in [1, k_i]$). We will also call components of a relation as *columns*, and the indexes $f_{i,j}$ as *labels* of columns. For technical reasons we only allow relations of different base sets: for all $i \in [1, n], j, k \in [1, k_i]$ such that $j \neq k$ the non-equality $f_{i,j} \neq f_{i,k}$ must hold. We may easily overcome this limitation by introducing new base sets.

We only deal with finite relations, so each A_i must be a finite set ($i = 1, \dots, m$). Since they are finite, we can encode them over a common alphabet X (e.g. the set of bits). Let these encodings denoted by the injective mappings $e_i : A_i \mapsto X^{l_i}$, where l_i is the letters needed to uniquely encode the elements of A_i in the alphabet X ($i \in [1, m]$). Please note that the length of $e_i(r)$ is always exactly l_i , independently of $r \in A_i$.

By the use of these mapping, we may define injective mappings from R_i to words over X . For each $i = 1, \dots, n$ define $h_i : R_i \mapsto X^{L_i}$, where $L_i = \sum_{j=1}^{k_i} l_{f_{i,j}}$, and for all $r = (r_1, \dots, r_{k_i}) \in R_i$, $h_i(r)$ is defined as the concatenation of the mappings of the components of r : $h_i(r) = e_{f_{i,1}}(r_1) \dots e_{f_{i,k_i}}(r_{k_i})$.

An element $r \in R_i$ ($i \in [1, n]$) is represented by a single linear complex over an alphabet Σ of length L_i : each letter is a triplet, where the first components are letters of X , such that the concatenation of the first components gives the word $h_i(r)$. The second components contain the index of the base set whose element is

partially encoded in the letter, and the last element is the position of the encoded letter within the encoding of the referred base set. Hence we define the alphabet

$$\Sigma = \{(x, j, k), (x, j, k)' \mid x \in X, j \in [1, m], k \in [1, l_j]\}$$

where each letter has its primed version as complementary pair and vice versa. Since the alphabet Σ is fixed, we have to define all relations occurring in the computation in advance. This problem can be eliminated if we use a suitable encoding of Σ . Now let us formally define for every $i \in [1, n]$ the injective mapping $g_i : X^{L_i} \mapsto \Sigma^{L_i}$ in the above manner: for every $x_1 \dots x_{L_i} \in X^{L_i}$

$$\begin{aligned} g_i(x_1 \dots x_{L_i}) = & (x_1, f_{i,1}, 1)(x_2, f_{i,1}, 2) \cdots (x_{l_{f_{i,1}}}, f_{i,1}, l_{f_{i,1}}) \\ & (x_{l_{f_{i,1}}+1}, f_{i,2}, 1) \cdots (x_{l_{f_{i,1}}+l_{f_{i,2}}}, f_{i,2}, l_{f_{i,2}})) \\ & \vdots \\ & (x_{L_i-l_{f_{i,k_i}}+1}, f_{i,k_i}, 1) \cdots (x_{L_i}, f_{i,k_i}, l_{f_{i,k_i}}). \end{aligned}$$

Define for every $i \in [1, n]$ the mapping $f_i = g_i \circ h_i$. Of course this mapping is also injective. An element of a relation $r \in R_i$ is represented by the single linear complex $(\{f_i(r)\}, \emptyset)$, and R_i is represented by the test tube containing the linear complex $(\{f_i(r) \mid r \in R_i\}, \emptyset)$ ($i \in [1, n]$).

Let us look at an example. We have one relation, $R_1 \subseteq A_1 \times A_2$, where $A_1 = \{i \mid 0 \leq i \leq 99\}$ and $A_2 = \{i \mid 0 \leq i \leq 9\}$. A possible encoding of the sets A_1 and A_2 is the decimal representation, we need two digits for A_1 and one digit for A_2 . We get: $X = \{0, 1, \dots, 9\}$, $l_1 = 2$, $l_2 = 1$. $h_1((a, b)) = a'b'$ where $a' \in X^2$, $b' \in X$, a' and b' are the decimal representation of a and b containing leading zeroes if necessary. The complex alphabet is the following:

$$\Sigma = \{(x, j, k), (x, j, k)' \mid x \in [0, 9], j \in [1, 2], (j = 1 \implies k \in [1, 2], j = 2 \implies k = 1)\}.$$

If $R_1 = \{(2, 3), (85, 0)\}$, then $h_1(R_1) = \{(02, 3), (85, 0)\}$, and

$$f_1(R_1) = \{(0, 1, 1)(2, 1, 2)(3, 2, 1), (8, 1, 1)(5, 1, 2)(0, 2, 1)\}.$$

The test tube of R_1 contains two single linear complexes, each containing one single string from $f_1(R_1)$.

Although the introduced representation is redundant, not too simple and have some limitations, we choose it because it is robust and allows simple implementation of the RA operations.

4 Operations

In this section we give methods for creating test tubes containing given relations as well as creating tubes from existing ones as a result of a RA operation. We

give a molecular program for Union, Selection, Cartesian product, Projection and Difference. The other RA operations can be expressed by these ones. In all our examples the test tube T_i will denote the tube that contains the representation of relation R_i ($i \in [1, n]$). We will use auxiliary tubes, too. These will be denoted by indexed S symbols.

Set up

After we fixed the originating relations and all computation by which we want to define new relations from the existing ones, we fix the alphabet Σ and the mappings defined in the previous section. The test tubes that represent the original relations can be set up by subsequent uses of the *Prepare* operation. The realization of this process in laboratory can be very expensive and time consuming for relations of many elements. An alternative method for creating large databases of DNA strands can be found in [3]. A third way can be starting from a naturally existent set of DNA strands and transform them to the form required in our model using biomolecular operations only.

Union

The execution program for creating the union R_k of two relations R_i and R_j is very simple ($i, j, k \in [1, n]$):

For $R_k := R_i \cup R_j$ do:

$S_1 := \text{Copy}(T_i)$

$S_2 := \text{Copy}(T_j)$

$\text{Merge}(S_1, S_2)$

$T_k := S_1$

We simply make copies of the tubes representing R_i and R_j , merge them together to form T_k .

Selection

We give different programs for the selection operation σ depending on the selection condition. First, suppose that the condition is that a given column equals to a constant value or formally: in the relation R_i the column labelled j is equal to $u \in A_j$, where $i \in [1, n]$, $j \in [1, m]$. Let the letters $x_1, \dots, x_{l_j} \in X$ be determined by the equation $e_i(u) = x_1 \dots x_{l_j}$.

For $R_k := \sigma_{c_j=u} R_i$ do:

$S_1 := \text{Copy}(T_i)$

$T_k := \text{Separate}_{\text{incl}}(S_1, (x_1, j, 1) \dots (x_{l_j}, j, l_j))$

The program is based on a single *Separate* instruction that selects from a copy of the original tube those strings, that contain the (possible long) encoding of word u

over alphabet Σ as subword. However, if for technical reasons we allow separations of short sequences only, we may take advantage of our redundant representation and perform the separation step by step, letter by letter, getting T_k after l_j *Separate* instructions:

For $R_k := \sigma_{c_j=u} R_i$ **do**:

$S_1 := \text{Copy}(T_i)$

$S_2 := \text{Separate}_{\text{incl}}(S_1, (x_1, j, 1))$

$S_3 := \text{Separate}_{\text{incl}}(S_2, (x_2, j, 2))$

\vdots

$S_{l_j} := \text{Separate}_{\text{incl}}(S_{l_j-1}, (x_{l_j-1}, j, l_j - 1))$

$T_k := \text{Separate}_{\text{incl}}(S_{l_j}, (x_{l_j}, j, l_j))$

Next, we show how to deal with selections where the condition is that two columns are equal. Select those elements of R_i whose columns labelled with a and b are equal. Let us suppose, that the two columns have a representation over X of equal length, that is suppose $l_a = l_b$.

For $R_k := \sigma_{c_a=c_b} R_i$ **do**:

$S_1 := \text{Copy}(T_i)$

For $j = 1, 2, \dots, l_a$ **do**

$S_2 := \emptyset$

For each $x \in X$ **do**

$S_3 := \text{Separate}_{\text{incl}}(S_1, (x, a, j))$

$S_4 := \text{Separate}_{\text{incl}}(S_3, (x, b, j))$

$\text{Merge}(S_2, S_4)$

end do

$S_1 := S_2$

end do

$T_k := S_1$

The inner loop separates those complexes of S , whose j th letter are equal both in column a and column b in the representation over X , since it is the union of such words, where the j th letter equal to an $x \in X$ in both columns for all $x \in X$. Having done this separation for all letters of the columns we get T_k , using a total of $2l_a |X|$ *Separate*, $l_a |X|$ *Merge*, $2l_a$ *Assign* and one *Copy* instructions.

If the condition contains \neq instead of $=$, we have to modify slightly the former algorithms to give the union of those complexes that differ in at least one position from the given constant value, or the value of the other column. If the condition contains the logical operator *and*, then we model it by successive selection. We model *or* by merging the resulting tubes of the constituent selections. The operation *not* can always be avoided using the former operators.

It is also possible to model selection operations that contain simple arithmetic expressions in their conditions. There are several methods, by which we can perform calculations on DNA molecules, see e.g. [6], [7] and [8]. However, dealing with

comparative relations $<$ and $>$ is not settled yet, to handle them is an open problem as of today.

Cartesian product

For creating the Cartesian product of two test tubes T_i and T_j we „stick” the proper ends of the strings in the tubes ($i, j \in [1, n]$). Because the result can have much more element, than the original relations, the test tubes must be amplified by a factor n , which is no smaller than the number of complexes in any of the two test tubes. An upper bound for n is of course $\max\{|R_i|, |R_j|\}$ as well as $\max\{|X|^{l_i}, |X|^{l_j}\}$. Please note that in our representation all column labels of a relation must be unique, so *before* executing the program creating the product of two tubes, we must *relabel* one of each pair of the columns that would have equal label in the product. This is especially important, if we take the Cartesian product of a relation with itself. The definition of relabelling and the molecular program for it is shown in the end of this section.

For $R_k := R_i \times R_j$ do:

```

 $S_1 := \text{Copy}(T_i)$ 
 $S_2 := \text{Copy}(T_j)$ 
 $\text{Amplify}(S_1, n)$ 
 $\text{Amplify}(S_2, n)$ 
 $S_3 := \text{Prepare}(\{(x, f_{j,1}, 1)'(y, f_{i,k_i}, l_{f_{i,k_i}})' \mid x, y \in X\})$ 
 $\text{Merge}(S_1, S_2)$ 
 $\text{Merge}(S_1, S_3)$ 
 $\text{Anneal}(S_1)$ 
 $\text{Ligate}(S_1)$ 
 $\text{Denature}(S_1)$ 
 $S_1 := \text{Select}_{=}(S_1, L_i + L_j)$ 
 $T_k := S_1$ 

```

We prepare a tube S_3 containing complexes of size two: the complements of any possible first symbol of R_j represented over the alphabet Σ followed by complements of any possible last symbol of R_i represented over the alphabet Σ . After merging this tube with the amplified copies of T_i and T_j , these complexes can anneal to the last letters of R_i and to the first letters of R_j . If both edges are annealed, the annealed complexes of R_i and R_j are ligated: they are stuck together, and remain stuck even when after *Denature*, the complex of S_3 breaks off. After these operations complexes of size $L_i + L_j$ form the elements of T_k representing the relation $R_k = R_i \times R_j$.

Projection

First let us show how can we project a relation R_i into a single column labelled j ($i \in [1, n], j \in [1, m]$):

For $R_k := \Pi_{c_j} R_i$ **do**:

$S_1 := \text{Copy}(T_i)$

for each $x \in X$ **do**

$\text{Cleave}_{\text{before}}(S_1, (x, j, 1))$

$\text{Cleave}_{\text{after}}(S_1, (x, j, l_{f_{i,j}}))$

end do

for each $x \in X$ **do**

$S_1 := \text{Separate}_{\text{incl}}(S_1, (x, j, 1))$

end do

$T_k := S_1$

We cleave (cut) the complexes *before* any possible first letter of the i th column represented in the alphabet Σ , and then cleave *after* any possible last letter. By that we cleave each complex into three parts: the parts (complexes) that contain any (let say, first) letter of the i th column will constitute the tube containing the projection.

When we want to project into more than one column, then in addition to cutting the unnecessary ends of the strings as in the previous case, we have to erase some inner „gaps” as well: substrings that do not belong to columns in the projection list, but laying between them. We will show a molecular program by which we can erase one gap. For modelling the general case of the projection we must call this procedure for all inner gaps, and than must cut the needless ends using a procedure very similar to the former one. Let us now look at the program that erases the gap between the a th and b th column in the tube representing the relation R_i ($i \in [1, n]$, $a, b \in [1, k_i]$, $a < b$):

For $S_0 := \text{EraseGap}(T_i, i, a, b)$ **do**:

$S_1 := \text{Prepare}(\{(x, f_{i,1}, 1)'(y, f_{i,k_i}, l_{f_{i,k_i}})' \mid x, y \in X\})$

$S_2 := \text{Prepare}(\{(x, f_{i,b}, 1)'(y, f_{i,a}, l_{f_{i,a}})' \mid x, y \in X\})$

$S_0 := \text{Copy}(T_1)$

$\text{Merge}(S_0, S_1)$

$\text{Anneal}(S_0)$

for each $x \in X$

$\text{Cleavage}_{\text{after}}(S_0, (x, f_{i,a}, l_{f_{i,a}}))$

$\text{Cleavage}_{\text{before}}(S_0, (x, f_{i,b}, 1))$

end do

$S_0 := \text{Select}_{=}(S_0, \sum_{j=1}^a l_{f_{i,j}} + \sum_{j=b}^{k_i} l_{f_{i,j}} + 2)$

$\text{Merge}(S_0, S_2)$

$\text{Anneal}(S_0)$

$\text{Ligate}(S_0)$

$\text{Denature}(S_0)$

$S_0 := \text{Select}_{=}(S_0, \sum_{j=1}^a l_{f_{i,j}} + \sum_{j=b}^{k_i} l_{f_{i,j}})$

We create the tube S_1 , whose complexes can anneal to the first and last letter of *any* complex in T_i . Using this tube we can achive that the strings of T_i form

loops, so that we can cut them with the possibility not to confuse the separated beginnings and endings. After creating the rings we cut the unneeded columns and stick the broken parts together with the help of the tube S_2 , whose complexes can anneal to the first letter of the b th column and to the last letter of the a th column. We may now select the result.

Difference

The last RA operation we examine is the set difference. For creating the test tube T_k that represents the difference of the relations represented in the tubes T_i and T_j we may use the following molecular program, which has a precondition that relations R_i and R_j has the same base sets ($i, j, k \in [1, n]$):

For $R_k := R_i \setminus R_j$ **do**:

```

 $S_1 := \text{Copy}(T_i)$ 
 $S_2 := \text{Copy}(T_j)$ 
 $S_3 := \text{Prepare}(\{(x, a, b)' \mid x \in X, a \in [1, m], b \in [1, l_a]\})$ 
 $\text{Amplify}(S_3, |R_j|)$ 
 $\text{Merge}(S_2, S_3)$ 
 $\text{Anneal}(S_2)$ 
 $\text{Ligate}(S_2)$ 
 $\text{Denature}(S_2)$ 
 $S_4 := \text{Select}_=(S_2, L_j)$ 
 $S_5 := \emptyset$ 
For each  $x \in X$  do
     $S_6 := \text{Separate}_{\text{incl}}(S_2, (x, f_{i,1}, 1)')$ 
     $\text{Merge}(S_5, S_6)$ 
end do
 $\text{Merge}(S_1, S_5)$ 
 $\text{Anneal}(S_1)$ 
 $S_7 := \text{Select}_=(S_1, L_i)$ 
 $S_8 := \emptyset$ 
For each  $x \in X$  do
     $S_9 := \text{Separate}_{\text{incl}}(S_7, (x, f_{i,1}, 1))$ 
     $\text{Merge}(S_8, S_9)$ 
end do
 $T_k := S_9$ 

```

After creating copies of T_i and T_j we create a tube S_3 , that contains complexes of size one: any possible primed letter of Σ . Note that this tube does not depend on T_i or T_j , it can be used for calculating other differences as well. After merging the suitably amplified tube of S_3 with the copy of T_j , the short complexes are permitted to anneal to the complexes of S_2 , hence after the *Anneal* and *Ligate* operations each complex of S_2 will be annealed (or paired) to an equally long complementary complex. After *Denature* these pairs of complexes break apart, and after some

selection and separation we get the tube S_5 , that contain strings that are exactly the complements of the strings in T_j . When we merge this tube with S_1 , those complexes that appear both in T_i and T_j will form pairs after annealing, and those complexes that appear only in one of the tubes T_i , or T_j remain in their linear structure. >From the result we only have to separate the linear complexes of T_i .

Relabelling

We say that the relation $R_j \subseteq A_{f_{j,1}} \times \dots \times A_{f_{j,k_j}}$ is a *relabelling* of $R_i \subseteq A_{f_{i,1}} \times \dots \times A_{f_{i,k_i}}$ if $k_i = k_j$, $R_i = R_j$ and for each $k \in [1, k_i]$ either $f_{i,k} = f_{j,k}$, or $f_{i,k} \neq f_{j,k}$, but $A_{f_{i,k}} = A_{f_{j,k}}$ and $e_{f_{i,k}} = e_{f_{j,k}}$. Hence the relabelled relation has the same base sets as the original relation, it has the same value, too, but some of its components may have a different label, but it does not affect the representation of that component over the alphabet X . Of course the representation of the two relations over the alphabet Σ will be different.

For our purposes it is enough show that relabelling where all of the columns are relabelled can be done in our model. An easy way of doing this is based on the fact that such relabelling can be expressed by relational operations:

$$R_j = \Pi_{f_{j,1}, \dots, f_{j,k_i}} \sigma_{f_{i,1}=f_{j,1} \wedge \dots \wedge f_{i,k_i}=f_{j,k_i}} R_i \times A_{f_{j,1}} \times \dots \times A_{f_{j,k_j}}.$$

The tube representing $A_{f_{j,1}} \times \dots \times A_{f_{j,k_j}}$ can effectively be prepared, in spite of the fact that this tube contains an exponential number of strings. After preparing the tube we may perform the marked operations as stated before. Another way for relabelling is to form a ring of each string (similarly to the *EraseGap* operation), cut each loop before and after the letter we want to replace, then bind the broken loops again inserting the substituting letter. After doing this for all letters of the columns to relabel, we are ready.

5 Conclusions

In this work we showed that building a relational database in test tubes using DNA strands is possible. The proof is based on the assertion that the RDNA model of biomolecular computing is indeed a sound model of biomolecular operations. It seems to be a correct model, because it is based on the basic structure of DNA strands and on the well understood operations on test tubes. However, by the time of writing no real laboratory experiments justified neither the model nor any application based on the model. During laboratory realization it may turn out, that alternate versions of the mentioned operations proves to be more efficient or reliable, it depends on the used laboratory techniques.

It is not unlikely in the not too far future, that we can make complex queries on artificially created or naturally existent DNA databases. Utilizing the enormous storage capacity of DNA and the possibility of associative searches on the strands it may be possible to efficiently work with databases of size much greater than that is manageable on conventional computer architectures.

References

- [1] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, November 11, 1994.
- [2] Eric B. Baum. Building an associative memory vastly larger than the brain. *Science*, 268:583–585, April 28, 1995.
- [3] John H. Reif, T. H. LaBean, M. Pirrug, V. S. Rana, B. Guo, C. Kingsford, and G. S. Wickham. Experimental construction of a very large scale DNA database with associative search capability. In *DNA Computing, 7th international Workshop on DNA-Based Computers, DNA 2001, Tampa, U.S.A., 10–13 June 2001*, pages 241–250. University of South Florida, 2001.
- [4] John H. Reif. Parallel molecular computation: Models and simulations. *Algorithmica*, 1998. Special issue on Computational Biology. See also [9].
- [5] Masanori Arita, Masami Hagiya, and Akira Suyama. Joining and rotating data with molecules. In *IEEE International Conference on Evolutionary Computation*, pages 243–248, Indiana University, Purdue University, Indianapolis, Illinois, April 13–16, 1997.
- [6] Pierluigi Frisco. Parallel arithmetic with splicing. *Romanian Journal of Information Science and Technology (ROMJIST)*, 3(2):113–128, 2000.
- [7] Eric B. Baum and Dan Boneh. Running dynamic programming algorithms on a DNA computer. In *Proceedings of the Second Annual Meeting on DNA Based Computers, held at Princeton University, June 10–12, 1996*. [10].
- [8] Leonard M. Adleman, Paul W. K. Rothmund, Sam Roweis, and Erik Winfree. On applying molecular computation to the data encryption standard. In *Proceedings of the Second Annual Meeting on DNA Based Computers, held at Princeton University, June 10–12, 1996*. [10].
- [9] John H. Reif. Parallel molecular computation: Models and simulations. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA95), Santa Barbara, June 1995*, pages 213–223. Association for Computing Machinery, June 1995. See also [4].
- [10] American Mathematical Society. *Proceedings of the Second Annual Meeting on DNA Based Computers, held at Princeton University, June 10–12, 1996.*, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science., 1996.

Various Hyperplane Classifiers Using Kernel Feature Spaces*

Kornél Kovács[†] and András Kocsor[‡]

Abstract

In this paper we introduce a new family of hyperplane classifiers. But, in contrast to Support Vector Machines (SVM) - where a constrained quadratic optimization is used - some of the proposed methods lead to the unconstrained minimization of convex functions while others merely require solving a linear system of equations. So that the efficiency of these methods could be checked, classification tests were conducted on standard databases. In our evaluation, classification results of SVM were of course used as a general point of reference, which we found were outperformed in many cases.

1 Introduction

Numerous scientific areas such as bioinformatics, pharmacology and artificial intelligence depend on classification and regression methods which may be linear or non-linear, but it now seems that by using the so-called kernel idea, linear methods can be readily generalized to nonlinear ones. The key idea was originally presented in Aizermann's paper [1] and it was successfully applied in the context of the ubiquitous Support Vector Machines [10]. The roots of SV methods can be traced back to the need for the determination of the optimal parameters of a separating hyperplane, which can be formulated both in input space or in kernel induced feature spaces. However, optimality can vary from method to method and SVM is just one of several possible approaches.

Without loss of generality we shall assume that, as a realization of multivariate random variables, there are m -dimensional real attribute vectors in a compact set \mathcal{X} over \mathbb{R}^m describing objects in a certain domain, and that we have a finite $n \times m$ sample matrix $X = [\mathbf{x}_1, \dots, \mathbf{x}_n]^T$ containing n random observations. Let us assume as well that we have an indicator function $\mathcal{L} : \mathbb{R}^m \rightarrow L \subseteq \mathbb{R}$, where $\mathcal{L}(\mathbf{x}_i)$ gives the label of the sample \mathbf{x}_i , and let us denote the vector $[\mathcal{L}(\mathbf{x}_1), \dots, \mathcal{L}(\mathbf{x}_n)]^T$ by $\mathcal{L}(X)$.

*This work was supported under the contract IKTA No. 2001/055 from the Hungarian Ministry of Education.

[†]Department of Informatics, University of Szeged H-6720 Szeged, Arpád tér 2., Hungary, e-mail: kkornel@inf.u-szeged.hu

[‡]Research Group on Artificial Intelligence of the Hungarian Academy of Sciences and University of Szeged, H-6720 Szeged, Aradi vértanúk tere 1., Hungary, e-mail: kocsor@inf.u-szeged.hu

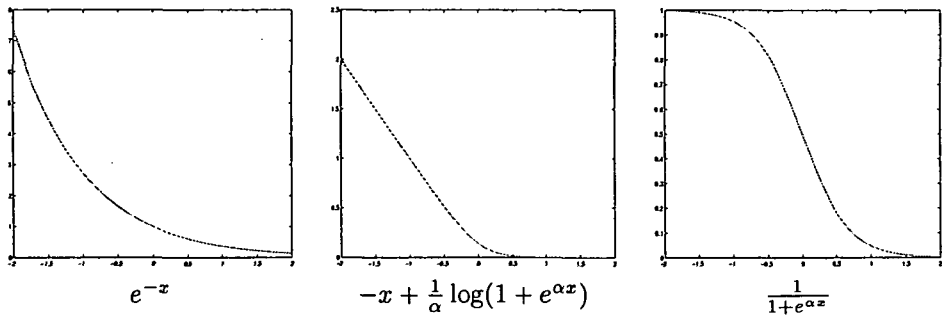


Figure 1: Three possible loss functions

Here, a finite set L means a classification task. Should L be an infinite set, the task will be a regression problem.

In this paper we will restrict our investigations only to that of binary classification ($L = \{-1, +1\}$), as multiclass problems can be dealt with by applying binary classifiers [3]. But regression problems will not be entirely excluded here, since binary classifiers will be derived from regression formulae.

2 Linear classifiers with various loss-functions

Linear classification attempts to separate the sample points with different labels via a hyperplane. A hyperplane is a set of point z :

$$(z^T \ 1) a = 0 \quad z \in \mathbb{R}^m, \quad a \in \mathbb{R}^{m+1}. \quad (1)$$

For a point z the left-hand side of Eq. (1) is a signed expression with absolute value proportional to the distance from the hyperplane. In addition, the sign of this expression corresponds to the sign of the half-space the point lies in.

A point x_i is well-separated by a hyperplane with parameter a if and only if:

$$\mathcal{L}(x_i) \cdot (x_i^T \ 1) a > 0 \quad i \in \{1, \dots, n\}.$$

Based on these products a target function - whose lower value indicates a better separation - can be defined:

$$\tau(a) = \sum_{i=1}^n g(\mathcal{L}(x_i) \cdot (x_i^T \ 1) a), \quad (2)$$

where $g: \mathbb{R} \rightarrow \mathbb{R}$ is a strictly monotonic decreasing function, called a loss function. Of the many possibilities [6], three candidates are shown in Fig. 1. We should note here that using a signum-function approximating loss function, the measure estimates the number of poorly separated points when $\alpha \rightarrow \infty$.

Minimizing $\tau(a)$ we get an unconstrained minimization of a strictly convex function, which is in marked contrast to the quadratic optimization with constraints

in SVM. With a suitably smooth loss function, the gradient vector of $\tau(\mathbf{a})$ will be smooth as well, hence one can apply quasi-Newton methods or even the Newton iteration method.

After obtaining the optimal parameter of the separating hyperplane the binary classification of an arbitrary point \mathbf{z} can be carried out by:

$$\text{sign}((\mathbf{z}^T \ 1) \mathbf{a}).$$

3 Linear regression in classification

Linear regression attempts to optimally fit a hyperplane onto the indicator function \mathcal{L} . The indicator function has values $\mathcal{L}(\mathbf{x}_1), \dots, \mathcal{L}(\mathbf{x}_n)$ at the sample points $\mathbf{x}_1, \dots, \mathbf{x}_n$ while the regression hyperplane has function values $f(\mathbf{x}_1), \dots, f(\mathbf{x}_n)$, where

$$f(\mathbf{z}) = (\mathbf{z}^T \ 1) \mathbf{a} \quad \mathbf{z} \in \mathbb{R}^m, \quad \mathbf{a} \in \mathbb{R}^{m+1}.$$

Thus the error of the sample point \mathbf{x}_i can be expressed by

$$\epsilon_i = \mathcal{L}(\mathbf{x}_i) - f(\mathbf{x}_i) = \mathcal{L}(\mathbf{x}_i) - (\mathbf{x}_i^T \ 1) \mathbf{a}.$$

The optimal parameter of the regression hyperplane can be obtained by minimizing the following sum:

$$\min_{\mathbf{a}} \sum_{i=1}^n \epsilon_i^2 = \min_{\mathbf{a}} \|\mathcal{L}(X) - X_1 \mathbf{a}\|_2^2 \quad X_1 = \begin{pmatrix} \mathbf{x}_1^T & 1 \\ \vdots & \vdots \\ \mathbf{x}_n^T & 1 \end{pmatrix},$$

whose well-known solution is given by

$$\mathbf{a} = (X_1^T X_1)^+ X_1^T \mathcal{L}(X), \quad (3)$$

where $^+$ denotes the Moore-Penrose pseudo-inverse.

Though the regression makes use of the hyperplane in a different sense from that in the classification problem, the regression-based binary classification of an arbitrary point \mathbf{z} can still be performed in the same way as that for a linear classifier:

$$\text{sign}((\mathbf{z}^T \ 1) \mathbf{a}).$$

4 Minor Component Classifier

Let us take the sample points X with the corresponding labels $\mathcal{L}(X)$, and represent $(\mathbf{x}_1^T, \mathcal{L}(\mathbf{x}_1))^T, \dots, (\mathbf{x}_n^T, \mathcal{L}(\mathbf{x}_n))^T$ as vectors in \mathbb{R}^{n+1} . In this extended space a hyperplane with parameter $\bar{\mathbf{a}}$ contains points \mathbf{z} where

$$(\mathbf{z}^T \ \mathcal{L}(\mathbf{z}) \ 1) \bar{\mathbf{a}} = 0, \quad \mathbf{z} \in \mathbb{R}^m, \quad \bar{\mathbf{a}} \in \mathbb{R}^{m+2}.$$

The distance of $(\mathbf{x}_i, \mathcal{L}(\mathbf{x}_i))$ from the hyperplane is

$$\delta(\mathbf{x}_i, \mathcal{L}(\mathbf{x}_i)) = \frac{(\mathbf{x}_i^T \quad \mathcal{L}(\mathbf{x}_i) \quad 1) \bar{\mathbf{a}}}{\|\bar{\mathbf{a}}\|_2},$$

so there exists an optimal hyperplane fitting on the extended sample points with least error:

$$\min_{\bar{\mathbf{a}}} \sum_{i=1}^n \delta(\mathbf{x}_i, \mathcal{L}(\mathbf{x}_i))^2 = \min_{\bar{\mathbf{a}}} \frac{\bar{\mathbf{a}}^T X_2^T X_2 \bar{\mathbf{a}}}{\bar{\mathbf{a}}^T \bar{\mathbf{a}}} \quad X_2 = \begin{pmatrix} \mathbf{x}_1^T & \mathcal{L}(\mathbf{x}_1) & 1 \\ \vdots & \vdots & \vdots \\ \mathbf{x}_n^T & \mathcal{L}(\mathbf{x}_n) & 1 \end{pmatrix}. \quad (4)$$

It can be proved that eigenvectors of $X_2^T X_2$ are the stationary points of the above functional with the corresponding eigenvalues as function values. Thus the solution of the minimization problem can be readily obtained by finding the eigenvector of $X_2^T X_2$ which has the smallest eigenvalue [4].

We should note that the better the fit of a hyperplane onto the points, the lower the deviation of the sample points projections onto the normal vector of the hyperplane. Finding the best hyperplane means performing a Minor Component Analysis (MCA) [5] in the extended space, as MCA searches for directions with a small deviation of the sample points projections.

The binary classification of a point \mathbf{u} in the original space can be performed by computing the absolute distances in the extended space for both labels $\{-1, 1\}$ and probabilities can be assigned to the labels via normalization:

$$P(\mathcal{L}(\mathbf{u}) = 1) = \frac{|\delta(\mathbf{u}, -1)|}{|\delta(\mathbf{u}, 1)| + |\delta(\mathbf{u}, -1)|}$$

$$P(\mathcal{L}(\mathbf{u}) = -1) = \frac{|\delta(\mathbf{u}, 1)|}{|\delta(\mathbf{u}, 1)| + |\delta(\mathbf{u}, -1)|}$$

5 Kernel-based nonlinearization

The proposed methods, linear classifiers, linear regression and minor component classifier performs linear separation in the original sample space. Making the separation nonlinear with kernels it must be shown that the methods optimal solutions are in the linear subspace of the appropriate extended points:

$$\mathbf{a} = X_1 \boldsymbol{\alpha} \quad \boldsymbol{\alpha} \in \mathbb{R}^n,$$

and

$$\bar{\mathbf{a}} = X_2 \boldsymbol{\beta} \quad \boldsymbol{\beta} \in \mathbb{R}^n.$$

Regarding a linear classifier the parameter vector \mathbf{a} can be decomposed into two perpendicular components \mathbf{a}_1 and \mathbf{a}_2 , where the first component lies in the subspace of the extended sample points X_1 :

$$\mathbf{a} = \mathbf{a}_1 + \mathbf{a}_2 \quad \mathbf{a}_1 = X_1 \boldsymbol{\alpha}, \quad \boldsymbol{\alpha} \in \mathbb{R}^n, \quad \mathbf{a}_1 \perp \mathbf{a}_2.$$

The form of the measure τ then becomes

$$\begin{aligned}\tau(\mathbf{a}) &= \sum_{i=1}^n g\left(\mathcal{L}(\mathbf{x}_i) \cdot \begin{pmatrix} \mathbf{x}_i^T & 1 \end{pmatrix} (\mathbf{a}_1 + \mathbf{a}_2)\right) = \\ &= \sum_{i=1}^n g\left(\mathcal{L}(\mathbf{x}_i) \cdot \begin{pmatrix} \mathbf{x}_i^T & 1 \end{pmatrix} \mathbf{a}_1 + \mathcal{L}(\mathbf{x}_i) \cdot \begin{pmatrix} \mathbf{x}_i^T & 1 \end{pmatrix} \mathbf{a}_2\right) = \\ &= \sum_{i=1}^n g\left(\mathcal{L}(\mathbf{x}_i) \cdot \begin{pmatrix} \mathbf{x}_i^T & 1 \end{pmatrix} \mathbf{a}_1\right),\end{aligned}$$

because \mathbf{a}_2 is orthogonal to all the extended sample points $\begin{pmatrix} \mathbf{x}_i^T & 1 \end{pmatrix}$.

Because the measure depends only on \mathbf{a}_1 , thus the minimization in fact can be performed in the linear subspace of the extended sample points X_1 . Actually, this result holds true for the other methods as well.

Utilizing the introduced formulas the solutions of the proposed methods can be found by optimizing α and β respectively:

$$\min_{\alpha} \sum_{i=1}^n g(\mathcal{L}(\mathbf{x}_i) \cdot \begin{pmatrix} \mathbf{x}_i^T & 1 \end{pmatrix} X_1^T \alpha), \quad (5)$$

$$\alpha = (X_1^T X_1 X_1 X_1^T)^+ X_1^T X_1 \mathcal{L}(X), \quad (6)$$

$$\min_{\beta} \frac{\beta^T X_2 X_2^T X_2 X_2^T \beta}{\beta^T X_2 X_2^T \beta}. \quad (7)$$

Supposing that the pairwise dot products of the extended sample points are known the above optimizations have some polynomial time complexity that depends on the sample points number. Since the time complexity of these methods is not a function of dimension, the original vectors can be transformed to a new space \mathcal{F} with $\phi: \mathcal{X} \rightarrow \mathcal{F}$ (see Fig. 2) where the separation can be achieved perhaps more effectively. Now let the dot product be implicitly defined by the kernel function κ in this finite or infinite dimensional feature space \mathcal{F} with the associated transformation ϕ :

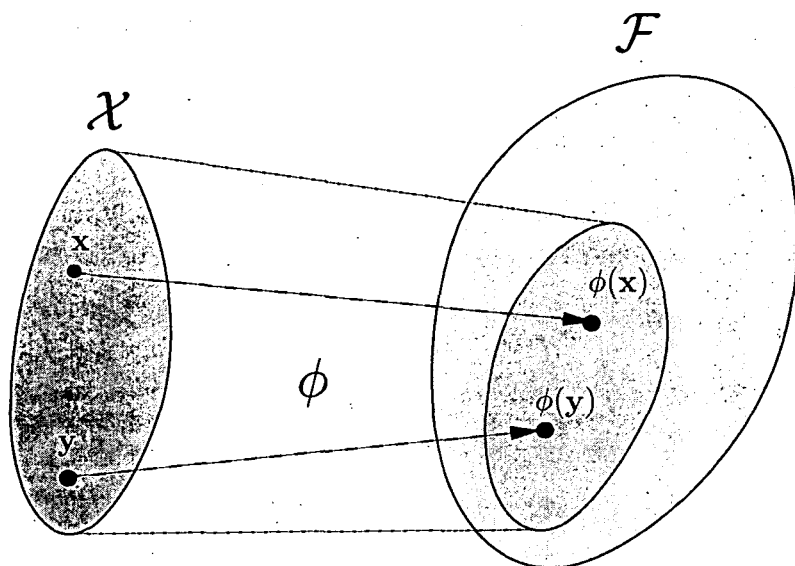
$$\kappa(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{y})$$

Algorithms using only the dot product can be executed in the kernel feature space by kernel function evaluations alone. Moreover, since ϕ is generally nonlinear the resultant method is nonlinear in the original sample space. Knowing ϕ explicitly - and, consequently, knowing \mathcal{F} - is not necessary. We need only define the kernel function, which then ensures an implicit evaluation. The construction of an appropriate kernel function (i.e. when such a function ϕ exists) is a non-trivial problem, but there are many good suggestions about the sorts of kernel functions [2, 7, 10] which might be adopted along with some background theory. Among the functions available, the two most popular kernels are:

$$\text{Polynomial kernel:} \quad \kappa(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T \mathbf{y} + 1)^d, \quad d \in \mathbb{N}$$

$$\text{Gaussian RBF kernel:} \quad \kappa(\mathbf{x}, \mathbf{y}) = e^{-\frac{\|\mathbf{x}-\mathbf{y}\|^2}{r}}, \quad r \in \mathbb{R}^+$$

For a given kernel function the dimension of the feature space \mathcal{F} is not always unique as in the case of a polynomial kernel, where it is at least $\binom{m+d-1}{d}$, while with the Gaussian RBF kernel we get an infinite dimension feature space.



$$\kappa(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{y})$$

Figure 2: The "kernel-idea". \mathcal{F} is the closure of the linear span of the mapped data. The dot product in the kernel feature space \mathcal{F} is defined implicitly. The dot product of $\phi(\mathbf{x})$ and $\phi(\mathbf{y})$ is $\kappa(\mathbf{x}, \mathbf{y})$.

Employing the kernel-idea to make the proposed methods (5), (6) and (7) non-linear, we obtain the following three expressions:

$$\min_{\alpha} \sum_{i=1}^n g \left(\mathcal{L}(\mathbf{x}_i) \cdot \sum_{j=1}^n \alpha_j \kappa((\mathbf{x}_i^T \ 1), (\mathbf{x}_j^T \ 1)) \right), \quad (8)$$

$$\alpha = (K^T K)^+ K^T \mathcal{L}(X), \quad (9)$$

$$\min_{\beta} \frac{\beta^T \bar{K} \bar{K} \beta}{\beta^T \bar{K} \beta}, \quad (10)$$

where the matrices K and \bar{K} contain the pairwise dot products of transformed points:

$$\begin{aligned} K_{ij} &= \kappa((\mathbf{x}_i^T \ 1), (\mathbf{x}_j^T \ 1)) \\ \bar{K}_{ij} &= \kappa((\mathbf{x}_i^T \ \mathcal{L}(\mathbf{x}_i) \ 1), (\mathbf{x}_j^T \ \mathcal{L}(\mathbf{x}_j) \ 1)). \end{aligned}$$

The solution of (10) can be obtained by finding the eigenvector corresponding to the smallest nontrivial eigenvalue of the generalized eigenproblem $\bar{K} \bar{K} \beta = \lambda \bar{K} \beta$.

Table 1: The best training and testing results using tenfold cross validations. A set of kernel functions with different parameters were used during the tests, but only the best results are summarized here.

	linear classifier	linear regression	MCC	SVM
BUPA	72.29	71.70	73.10	72.40
	65.98	65.40	62.24	65.60
chess	100.0	97.42	95.98	100.0
	98.08	90.73	88.49	98.08
echo	100.0	92.35	91.57	100.0
	89.54	89.57	90.32	90.10
hheart	86.64	85.96	85.27	87.10
	80.08	79.73	80.40	80.40
monks	100.0	93.35	93.35	100.0
	87.88	88.81	89.60	89.10
spiral	100.0	100.0	100.0	100.0
	88.48	87.23	90.80	89.20

Note here that if the transformed sample points lies entirely on a hyperplane in the space \mathcal{F} then the normal vector of the hyperplane is not in the subspace of the transformed sample points. Thus perfect fitting of the hyperplane is never realized in regression methods nonlinearized with kernels.

6 Experimental Results and Evaluation

When evaluating the efficiency of a new algorithm the usual method is to assess its performance by making use of standard databases. To this end we selected a set of databases from the UCI Repository [9]. Namely, we carried out tests using the BUPA liver, chess, echo, Hungarian heart, monks and spiral databases. All sets were normalized so that each feature had a zero mean and unit deviation and we applied a tenfold cross-validation on all the sets. Since a recent study [3] compared five different Support Vector algorithms using the UCI Repository and concluded that the methods have no significant difference in efficiency, we will employ [8] as the SVM classifier. The numerical results of tenfold cross-validations are shown in Table 1, where the best result is emphasized in bold. It confirms that regression based classification methods are indeed just as effective as the original separation algorithms. Moreover, making use of the labels in the regression task with the Minor Component Classifier the usual classification methods were surpassed in many cases so MCC can now be considered as a rival classification method.

References

- [1] M. AIZERMANN, E. BRAVERMAN, AND L. ROZONOER *Theoretical foundations of the potential function method in pattern recognition learning*, Automation and Remote Control 25:821-837, 1964.
- [2] CRISTIANINI, N. AND SHAW-TEY, J. *An Introduction to Support Vector Machines and other kernel-based learning methods*, Cambridge University Press, 2000.
- [3] HSU, C.-W. AND LIN, C.-J. *A comparison of methods for multi-class support vector machines*, IEEE Transactions on Neural Networks, Vol. 13, pp. 415-425, 2002.
- [4] FUHRMANN, D. R. AND LIU, B. *An iterative algorithm for locating the minimal eigenvector of a symmetric matrix*, Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing, pp. 45.8.1-45.8.4, 1984.
- [5] LUO, F., UNBEHAUEN, R., CICHOCKI, A. *A minor component analysis algorithm*, Neural Networks, Vol. 10/2, pp. 291-297, 1997.
- [6] EVGENIOU, T., PONTIL, M., POGGIO, T. *Regularization Networks and Support Vector Machines*, Advances in Computational Mathematics, Vol. 13/1, pp. 1-50, 2000.
- [7] SMOLA, A., BARTLETT, P., SCHÖLKOPF, B., SCHUURMANS, D. *Advances in Large Margin Classifiers*, MIT Press, Cambridge, 2000.
- [8] COLLOBERT, R. AND BENGIO, S. *SVM-Torch: Support Vector Machines for Large-Scale Regression Problems*, Journal of Machine Learning Research, vol 1, pages 143-160, 2001.
- [9] BLAKE, C. L. AND MERZ, C. J. *UCI repository of machine learning databases*, <http://www.ics.uci.edu/mlearn/MLRepository.html>, 1998.
- [10] VAPNIK, V. N. *Statistical Learning Theory*, John Wiley & Sons Inc., 1998.

Probabilistic Diagnostics with P-Graphs*

Balázs Polgár[†] and Endre Selényi[†]

Abstract

This paper presents a novel approach for solving the probabilistic diagnosis problem in multiprocessor systems. The main idea of the algorithm is based on the reformulation of the diagnostic procedure as a P-graph model. The same, well-elaborated mathematical paradigm—originally used to model material flow—can be applied in our approach to model information flow. This idea is illustrated by deriving a maximum likelihood diagnostic decision procedure. The diagnostic accuracy of the solution is considered on the basis of simulation measurements, and a method of constructing a general framework for different aspects of a complex problem is demonstrated with the use of P-graph models.

Introduction

Diagnostics is one of the major tools for assuring the reliability of complex systems in information technology.

In such systems the test process is often implemented on system-level: the “intelligent” components of the system test their local environment and each other. The test results are collected, and based on this information the good or faulty state of each system-component is determined. This classification procedure is known as *diagnostic process*.

The early approaches that solve the diagnostic problem employed oversimplified binary fault models, could only describe homogeneous systems, and assumed the faults to be permanent. Since these conditions proved to be impractical, lately much effort has been put into extending the limitations of traditional models [1]. However, the presented solutions mostly concentrated on only one aspect of the problem. In this paper we introduce a novel modeling approach based on P-graphs that can integrate these extensions in one framework, while maintaining a good diagnostic performance. With this model, we formulate diagnosis as an optimization problem and apply the idea to the well-known multiprocessor testing problem, whose structure is one of the simplest.

*This research has been supported partly by the Hungarian National Research Foundation Grants OTKA T038027.

[†]Department of Measurement and Information Systems, Budapest University of Technology and Economics, Magyar Tudósok krt. 2, Budapest, Hungary, H-1117, e-mail: {polgar,selenyi}@mit.bme.hu

Furthermore, we have not only integrated existing solution methods, but proceeding from a more general base we have extended the set of solvable problems with new ones.

The paper is structured as follows. First an overview is given about the traditional aspects of system-level diagnosis and the way we have generalized the test invalidation model. Then the elements and the solution method of a P-graph model are introduced. In the main part the diagnostic problem of a multiprocessor system is formulated with the use of P-graphs. Afterwards, an important aspect, the extensibility of the model is demonstrated via examples. Moreover, the generation and the solution method of a P-graph model is clarified on a small example. The diagnostic accuracy of the decoding algorithm is presented on the basis of simulation results and it is compared to other approaches taken from the literature. Finally, we conclude and sketch the direction of future work.

1 System-level Diagnosis

System-level diagnosis considers the *replaceable units* of a system, and does not deal with the exact location of faults within these units. A *system* consists of an interconnected network of independent but cooperating *units* (typically processors). The state of each unit is either *good* when it behaves as specified, or *faulty*, otherwise. The *fault pattern* is the collection of the states of all units in the system. A unit may test the *neighboring* units connected with it via direct links. The network of the units testing each other determines the *test topology*. The outcome of a test can be either *passed* or *failed* (denoted by 0/1 or G/F); this result is considered *valid* if it corresponds to the actual physical state of the tested unit.

The collection of the results of every completed test is called the *syndrome*. The test topology and the syndrome are represented graphically by the *testing graph*. The vertices of a testing graph denote the units of the system, while the directed arcs represent the tests originated at the *tester* and directed towards the *tested unit* (UUT). The result of a test is shown as the label of the corresponding arc. Label 0 represents the passed test result, while label 1 represents the failed one. See Figure 1 for an example testing graph with three units.

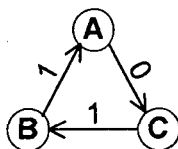


Figure 1: Example testing graph (test topology with syndrome)

1.1 Traditional approach

Traditional diagnostic algorithms [2, 3] assume that

- faults are permanent,
- states of units are binary (*good*, *faulty*),
- the test results of good units are always valid,
- the test results of faulty units can also be invalid. The behavior of faulty tester units is expressed in the form of *test invalidation models*.

Table 1 covers the possible test invalidation models where the selection of c and d values determines a specific model. The most widely used example is the so-called PMC (Preparata, Metze, Chien) test invalidation model, ($c = \text{any}$, $d = \text{any}$) which considers the test result of a faulty tester to be independent of the state of the tested unit. Another well-known test invalidation model is the BGM (Barsi, Grandoni, Maestrini) model ($c = \text{any}$, $d = \text{faulty}$) where a faulty tester will always detect the failure of the tested unit, as it is assumed that the probability of two units failing the same way is negligible.

Table 1: Traditional test invalidation models

State of tester	State of UUT	Test result
<i>good</i>	<i>good</i>	<i>passed</i>
<i>good</i>	<i>faulty</i>	<i>failed</i>
<i>faulty</i>	<i>good</i>	$c \in \{\text{passed}, \text{failed}, \text{any}\}$
<i>faulty</i>	<i>faulty</i>	$d \in \{\text{passed}, \text{failed}, \text{any}\}$

The purpose of system-level diagnostic algorithms is to determine the state of each unit from the syndrome. The difficulty comes from the possibility that a fault in the tester processor invalidates the test result. As a consequence, multiple “candidate” diagnoses can be compatible with the syndrome. To provide a complete diagnosis and to select from the candidate diagnoses, the so-called *deterministic* algorithms use extra information in addition to the syndrome, such as assumptions on the size or on the topology of the fault pattern.

Alternatively, *probabilistic* algorithms try to determine the most probable diagnosis assuming that a unit is more likely good than faulty [4]. Frequently, this maximum likelihood strategy can be expressed simply as “many faults occur less frequently than a few faults.” Thus, the aim of diagnostics is to determine the minimal set of faulty elements of the system that is consistent with the syndrome.

1.2 Generalized approach

In our previous work [5] we used a generalized test invalidation model, introduced by Blount [6]. In this model probabilities are assigned to both possible test outcome

for each combination of the states of tester and tested unit (shown in Table 2). Since the passed and failed results are complementary events, the sum of the probabilities in each row is 1. The assumption of the complete fault coverage can be relaxed in the generalized model by setting probability p_{b1} to the fault coverage of the test. Probabilities p_{c0} , p_{c1} , p_{d0} and p_{d1} express the distortion of the test results by a faulty tester. Moreover, the generalized model is able to encompass *false alarms* (a good tester finds a good unit to be faulty) by setting probability p_{a1} to nonzero.

Table 2: Generalized testing model

State of tester	State of UUT	Probability of test result	
		0	1
good	good	p_{a0}	p_{a1}
good	faulty	p_{b0}	p_{b1}
faulty	good	p_{c0}	p_{c1}
faulty	faulty	p_{d0}	p_{d1}

Naturally, the generalized test invalidation model also covers the traditional models. Setting the probabilities as $p_{a0} = p_{b1} = 1$, $p_{c0} = p_{c1} = p_{d0} = p_{d1} = 0.5$, and $p_{a1} = p_{b0} = 0$, the generalized model will have the characteristics of the PMC model, while the configuration $p_{a0} = p_{b1} = p_{d1} = 1$, $p_{c0} = p_{c1} = 0.5$ and $p_{a1} = p_{b0} = p_{d0} = 0$ will make it behave like the BGM model. Analogously, every traditional test invalidation model can be mapped as a special case to our model by assigning suitable probabilities to each element of the related test invalidation relation. In this sense the generalized test invalidation model covers the traditional models.

2 Diagnosis Based on P-Graphs

2.1 Definition of P-Graph Model of the Diagnostic System

The name 'P-graph' originates from the name 'Process-graph' from the field of Process Network Synthesis problems (PNS problem for short) in chemical engineering. In connection with this field the mathematical background of the solution methods of PNS problems have been elaborated well, see [7], [8] and [9].

A P-graph is a directed bipartite graph. Its vertices are partitioned into two sets, with no two vertices of the same set being adjacent. In our interpretation one of the sets contains *knowledge* (the knowledge about the states of units union the knowledge about the possible test results), the other one contains *logical relations* between the pieces of knowledge. The edges of the graph point from the *premisses*¹ 'through' the logical relation to the *consequences*. The set of premisses contains both good and faulty states of each unit (e.g., 'unit A is good', 'unit A is faulty', 'unit B is good', denoted by A_g , A_f , B_g), and the set of consequences contains the

¹premiss = preliminary condition

measured test results (e.g. 'unit A finds unit B to be good', 'unit B finds unit C to be faulty', denoted by AB_G , BC_F). Logical relations determine the possible premisses of each possible test result. Namely, there are 8 logical relations for each test according to the states of tester and tested unit and the possible test results. Probabilities in Table 2 are assigned to relations expressing the uncertainty of the consequences, see Figure 2.

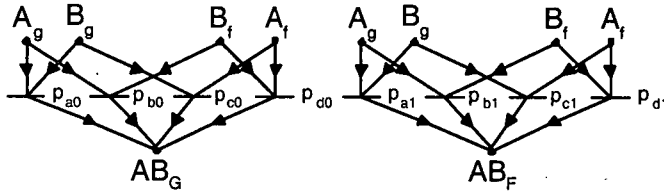


Figure 2: P-graph model of a single test (vertices with same label represent a single vertex; multiple instances are only for better arrangement)

A *solution structure* is defined as a subgraph of the original P-graph, which deduces the consequences back to a subset of premisses.

Function $X()$ is a membership function, $X(A)$ is 1 if unit A is in the solution structure, and 0 otherwise. With the use of this function *constraints* can be defined assuring that in a solution structure a unit should have one and only one state. Formally, for each unit U $X(U_G) + X(U_F) = 1$. A P-graph is *contradictionless* if all constraints are satisfied.

The *probability of the syndrome* (P_S) is the product of probabilities of relations in a solution structure. This is the occurring probability of the known consequences under the conditions of the given subset of system premisses.

Because of probabilities are assigned to relations, more contradictionless solution structures can exist having different subsets of system premisses and having different P_S values. The object is to find a solution structure containing that subset of system premisses which implies the known consequences with the maximum likelihood. This is an optimization task.

In principle, this task can be solved by general mathematical programming methods like mixed integer non-linear programming (MINLP), however, they are unnecessary complex. Friedler et al. ([7, 8, 9]) developed a new framework for solving PNS problems effectively by exploiting the special structure of the problem and the corresponding mathematical model.

2.2 Steps of the Solution Algorithm

1. The maximal P-graph structure is generated. It contains only the relevant pieces of knowledge and the relevant logical relations, but constraints are not yet satisfied. It contains all possible fault patterns being consistent with the given syndrome.

2. Every combinatorially feasible solution structure is obtained. These are the structures that satisfy the constraints and draw the known consequences—the syndrome—back to a subset of the system premisses. Each of these subsets determines a possible fault pattern.
3. For each combinatorially feasible solution structure the probability of syndrome is calculated. This is the conditional probability of the syndrome under the condition of a particular fault pattern.
4. The structure having the highest probability is selected; this solution structure contains the diagnosis with maximum likelihood.

Steps 2–4 can be completed either by a general solver for linear programming (since the generated maximal structure is a special flat P-graph), or with an adapted SSG algorithm [7] using the branch and bound technique.

3 Extensions of the Model

The main contribution of this novel modeling approach is its generality. With its use several aspects of system-level diagnosis can be handled in the same framework. Furthermore, it also became possible to formulate new aspects of diagnosis. So, it is possible to model and diagnose for instance

- *systems with heterogeneous elements*

To achieve this, different generalized test invalidation models with appropriate probabilities should be assigned to units with different behavior.

- *multiple fault states*

It is able to construct and handle a finer model of the state of a unit, than the binary one (containing the *good* and *faulty* states). This also means that the result of a test can be more than binary.

- *intermittent faults*

These are permanent faults that become activated only in special circumstances. Because these circumstances are usually independent from the testing process, these type of faults are diagnosed on the basis of multiple syndromes.

- *failures occurring during the test process*

It is a new aspect of the diagnostics. Traditional models all have the restrictive assumption that the state of units must be unchanged from the beginning of the test process to the end. But it is not acceptable if the time of test is comparable to the mean time between failures.

The model of the last two items are presented in details in the next subsections. The model constructed for the test process of intermittent faults is equivalent to the model of a system having more than two possible test results. Accordingly, for details of the handle of multiple fault states see the handle of intermittent faults.

3.1 Modeling Intermittent Faults

Although handle of intermittent faults is one of the difficult to manage diagnostic problems, a possible solution is the use of multiple syndromes, as mentioned above. In this approach two or more testing rounds are performed in a row, and the possible differences between the subsequent syndromes are used to detect intermittent faults.

The adaptation of diagnostic P-graph model to this approach is quite simple. Considering the case of double syndromes (for simplicity), there are four possible result combinations for each test:

- 'both results are passed' (denoted by GG),
- 'first result is passed, the other is failed' (denoted by GF),
- 'first result is failed, the other is passed' (denoted by FG) and
- 'both results are failed' (denoted by FF).

This means that the P-graph model of a single test should contain 4×4 logical relations according to the 4 possible test results and the 4 possible state-combinations of tester and tested unit (Figure 3). The probabilities of relations are calculated from the original probabilities (Table 2) and can be seen in Table 3.

Table 3: Probabilities of test results for pairs of syndrome

State of tester	State of UUT	Probability of test results			
		GG	GF	FG	FF
good	good	$p_{A0} = p_{a0}^2$	$p_{A1} = p_{a0}p_{a1}$	$p_{A2} = p_{a1}p_{a0}$	$p_{A3} = p_{a1}^2$
good	faulty	$p_{B0} = p_{b0}^2$	$p_{B1} = p_{b0}p_{b1}$	$p_{B2} = p_{b1}p_{b0}$	$p_{B3} = p_{b1}^2$
faulty	good	$p_{C0} = p_{c0}^2$	$p_{C1} = p_{c0}p_{c1}$	$p_{C2} = p_{c1}p_{c0}$	$p_{C3} = p_{c1}^2$
faulty	faulty	$p_{D0} = p_{d0}^2$	$p_{D1} = p_{d0}p_{d1}$	$p_{D2} = p_{d1}p_{d0}$	$p_{D3} = p_{d1}^2$

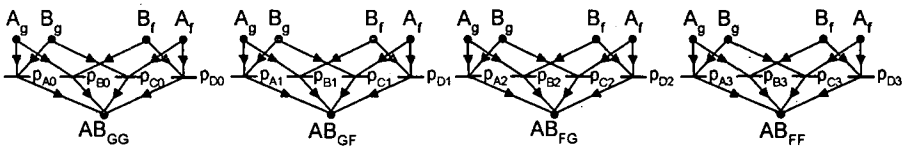


Figure 3: P-graph model of a single test in case of two syndromes

The case of diagnostics on the basis of more than two syndromes can be handled a similar way having more and more test result combinations.

A good property of this model is the following: after the 1st solution step—namely cutting the irrelevant parts of the graph to be solved—the P-graph model based on multiple syndromes is exactly of the same size as the P-graph model based on a single syndrome. This is because only the number of possible test results (or result combinations) grows, but the measured result (or result combination) of a test will be always a particular one.

3.2 Modeling Failures Occurring During the Test Process

Properties of the system to be modelled:

- faults are still permanent,
- units can fail during test process, i.e. a unit which was assumed to be good in a test can be faulty later. (Repairing is not included in the model, that is a faulty unit cannot become good in a later test.)

The second property implies that an order between tests should be defined and the states of a unit in different tests should be distinguished.

Let's define the *test order graph* $TO(V_{TO}, E_{TO})$, where

- each $t_i \in V_{TO}$ vertex represents a test in the system, i.e. it corresponds to an edge in the testing graph
- a $(t_i, t_j) \in E_{TO}$ directed edge defines a *preceding relation* between tests meaning that test t_i is performed earlier than test t_j .

For instance, consider a system with toroidal mesh topology, where each unit tests its four neighbors (Figure 4.a). The TO-graph of this system can be seen on Figure 4.b if only the order of those tests are known, which are performed by the same tester.

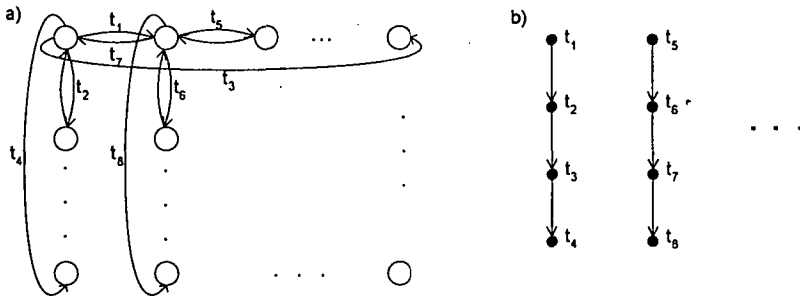


Figure 4: Example a) testing graph with toroidal mesh topology b) a possible test order graph of it

The definition of the P-graph model corresponds to the former one (Section 2.1) with the following changes.

- The set of *system premisses* contains each possible state of each unit in each such test, where the given unit is affected (for each U unit and t_i test U_iG and U_iF are included, where unit U is either a tester or a tested unit in test t_i).
- *Constraints* formulate that
 - each unit U in each test t_i where it is either a tester or a tested unit has one and only one state, i.e. $X(U_iG) + X(U_iF) = 1$.
 - for each unit U and tests t_i, t_j , where unit U is either a tester or a tested unit in tests t_i and t_j , and there exists a directed path from t_i to t_j in the TO-graph: $X(U_iF) + X(U_jG) \leq 1$.

Expectedly, the more information known about the dependencies of tests results the more accurate diagnosis. And reversely, the less edges in the test order graph can imply the more and more misdiagnosed processor in the diagnosis.

4 Example

Consider the testing graph and syndrome given on Figure 1. Eight logical relations belong to each of the three tests, but the maximal structure contains only four for each test depending on the test results as Figure 5 shows.

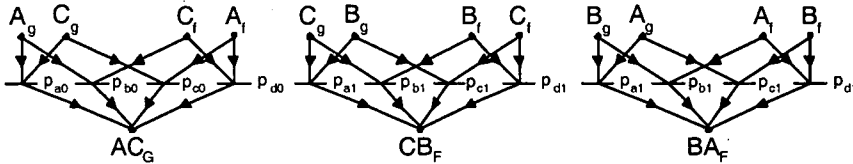


Figure 5: P-graph-model of testing graph and syndrome given on Figure 1

Eight combinatorially feasible solution structures exist because of the constraints and each of it contains three logical relations. The eight structures correspond to the 2^3 possible fault patterns of the three units. A part of these can be seen on Figure 6 with the corresponding diagnoses and probabilities. Finally, such a fault pattern is selected, which produces the syndrome with the highest probability.

Table 4 contains three test invalidation models, the first one corresponds to the PMC model, the second is a PMC model with incomplete fault coverage and the third is a more general model converging to the BGM model. The conditional probabilities of the syndrome under the conditions of different fault patterns, that is the redundant probabilities of the structures can be found in Table 5.

In case of PMC model the probability of syndrome is the highest when only unit B is faulty. This is still the case when the assumption of 100 percent test coverage

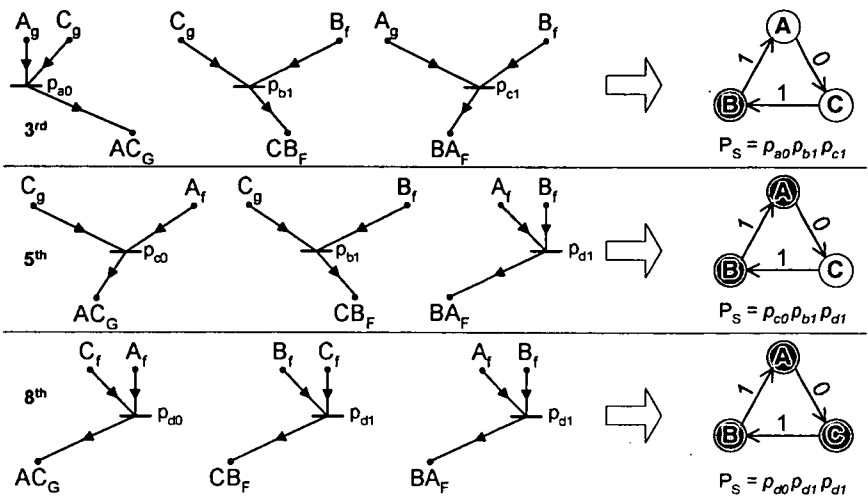


Figure 6: Some of solution structures of the P-graph model

Table 4: Test invalidation models with different probabilities

State of tester	State of tested unit	Test result					
		PMC		incomplete PMC		incomplete BGM-like	
		0	1	0	1	0	1
good	good	1	0	1	0	1	0
good	faulty	0	1	0.1	0.9	0.1	0.9
faulty	good	0.5	0.5	0.5	0.5	0.7	0.3
faulty	faulty	0.5	0.5	0.5	0.5	0.1	0.9

is given up but with smaller probability and with the possibility that unit C can be faulty although a good unit tested it to be good. If we assume that faults in the testers eventuate in valid test results more frequently than in invalid ones—as in the third model—then logically it seems to be probable that unit A is also faulty beside unit B and the algorithm provides this diagnosis.

Table 5: Probabilities of the syndrome (P_S) assuming different fault patterns and test invalidation models

Solution	1 st	2 nd	3 rd	4 th	5 th	6 th	7 th	8 th
Faulty units		A	B	C	A B	A C	B C	A B C
PMC	0	0	0.5	0	0.25	0.25	0	0.125
inc. PMC	0	0	0.45	0	0.225	0.225	0.025	0.125
inc. BGM-like	0	0	0.27	0	0.567	0.027	0.027	0.081

5 Simulation Results

In order to measure the efficiency of the P-graph based modeling technique a simulation environment was developed, which generates the fault pattern and the corresponding syndrome for the most common topologies with various parameters. The P-graph model of the syndrome-decoding problem was solved as a linear programming task using a commercial program called CPLEX. Other diagnostic algorithms with different solution methods taken from the literature were also implemented for comparison. First, the accuracy of the developed algorithm is demonstrated for varying parameters, then its relation to other algorithms for fixed parameters.

The simulations were performed in a two-dimensional toroidal mesh topology, where each unit is tested by its four neighbors and each unit behaved according to the PMC test invalidation model. Statistical values were calculated on the basis of 100 diagnostic rounds. In every round the fault pattern was generated by setting each processor to be faulty with a given probability, independently from others.

Accuracy of the solution algorithm: measurements were performed with system sizes of 4×4 , 6×6 , 8×8 , 10×10 units, and the failure probability of units varied from 10% to 100% in 10% steps. From the diagrams in Figure 7 it can be observed that the algorithm has a very good diagnostic accuracy. Even if half of the units were faulty, the rate of rounds containing misdiagnosed units did not exceed 20 percent, and the rate of misdiagnosed units relative to the system size was under 1 percent.

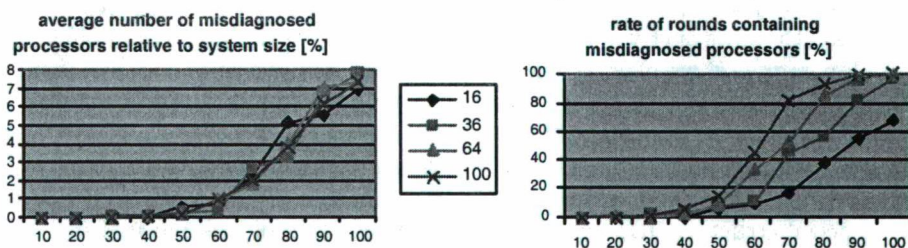


Figure 7: Simulation results depending on failing probability of units

Comparison to other algorithms: measurements were performed with system size 8×8 and the unit failure probability varied from 10% to 100% in 10% steps. The well-known algorithms taken from the literature were the LDA1 algorithm of Somani and Agarwal [10], the Dahbura, Sabnani and King (DSK) algorithm [11], and the limited multiplication of inference matrix (LMIM) algorithm developed by Bartha and Selenyi [12] from the area of local information diagnosis. It can be seen on the diagrams in Figure 8 that only the LMIM- algorithm approximates the accuracy of P-graph-algorithm.

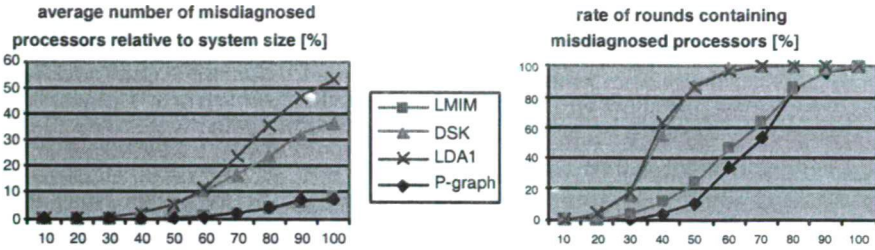


Figure 8: Comparison of probabilistic diagnostic algorithms

6 Conclusions

Application of P-graph based modeling in system-level diagnosis provides a general framework that supports the solution for several different fields, which previously needed several different modeling approaches and solution algorithms. Because the P-graph model takes into consideration more properties of the real system than previous models, its diagnostic accuracy is also better; it provides almost good diagnosis still in the situation, when half of the processors are faulty.

The results presented in this paper arose from solving the model with a general LP-problem solver and not from solving with a method specialized for PNS problems. Therefore its complexity was incomparable with traditional ones. But combinatorial approach for solving PNS problems is based on rigorous mathematical foundation, which –on the basis of experiences– can result in effective solution algorithm. Creating such an adapted algorithm is one of the subjects of our future work. Furthermore, we plan to examine the P-graph model of a diagnostic system with transient faults.

The favorable properties of the approach are achieved by considering the diagnostic system as a structured set of knowledge with well-defined relations. As mentioned previously, the syndrome-decoding problem in multiprocessor systems has a special structure, namely the direct manifestation of internal fault states in the syndromes. In more complex systems the states of the control logic have to be taken into account in the model to be analyzed [13]. These straightforward extensions to the modelling of integrated diagnostics can be well incorporated into the P-graph based models. Our current work aims at generalization of the results into this direction by extending previous results on the qualitative modeling of dependable systems with quantitative optimization [14].

References

- [1] T. Bartha, E. Selényi. Probabilistic System-Level Fault Diagnostic Algorithms for Multiprocessors, *Parallel Computing*, vol. 22, no. 13, pp. 1807–1821, Elsevier Science, 1997.

- [2] T. Bartha. Efficient System-Level Fault Diagnosis of Large Multiprocessor Systems, *Ph.D thesis*, BME-MIT, 2000.
- [3] M. Barborak, M. Malek, and A. Dahbura. The Consensus Problem in Fault Tolerant Computing, *ACM Computing Surveys*, vol. 25, pp. 171–220, June 1993.
- [4] S. N. Maheshwari, S. L. Hakimi. On Models for Diagnosable Systems and Probabilistic Fault Diagnosis. *IEEE Transactions on Computers*, vol. C-25, pp. 228–236, 1976.
- [5] B. Polgár, Sz. Nováki, A. Pataricza, F. Friedler. A Process-Graph Based Formulation of the Syndrome-Decoding Problem, In *4th Workshop on Design and Diagnostics of Electronic Circuits and Systems*, pp. 267–272, Hungary, 2001.
- [6] M. L. Blount. Probabilistic Treatment of Diagnosis in Digital Systems, In *Proc. of 7th IEEE International Symposium on Fault-Tolerant Computing (FTCS-7)*, pp. 72–77, June 1977.
- [7] F. Friedler, K. Tarjan, Y. W. Huang, L. T. Fan. Combinatorial Algorithms for Process Synthesis. *Comp. in Chemical Engineering*, vol. 16, pp. 313–320, 1992.
- [8] F. Friedler, K. Tarjan, Y. W. Huang, and L. T. Fan. Graph-Theoretic Approach to Process Synthesis: Axioms and Theorems, *Chemical Engineering Science*, 47(8), pp. 1973–1988, 1992.
- [9] F. Friedler, L. T. Fan, and B. Imreh. Process Network Synthesis: Problem Definition. *Networks*, 28(2), pp. 119–124, 1998.
- [10] A. Somani, V. Agarwal. Distributed syndrome decoding for regular interconnected structures, In *19th IEEE International Symposium on Fault Tolerant Computing*, pp. 70–77, IEEE 1989.
- [11] A. Dahbura, K. Sabnani, and L. King. The Comparison Approach to Multiprocessor Fault Diagnosis, *IEEE Transactions on Computers*, vol. C-36, pp. 373–378, Mar. 1987.
- [12] T. Bartha, E. Selényi. Probabilistic Fault Diagnosis in Large, Heterogeneous Computing Systems, *Periodica Polytechnica*, vol. 43/2, pp. 127–149, 2000.
- [13] A. Pataricza. Algebraic Modelling of Diagnostic Problems in HW-SW Co-Design. In *Digest of Abstracts of the IEEE International Workshop on Embedded Fault-Tolerant Systems*. Dallas, Texas, Sept. 1996.
- [14] A. Pataricza. Semi-decisions in the validation of dependable systems In *Proc. of IEEE International Conference on Dependable Systems and Networks*, pp. 114–115, Göteborg, Sweden, July 2001.

Programming by steps

Raluca Oana Scarlatescu*

Abstract

The paper introduces a new method of software analysis, design and programming based on a different implementation of a logical flow: the sequence of steps is memorised in a database table, and each step is linked to a specific function inserted in a library. A main application manages the steps' information and runs the functions, until the steps are finished. The database management system stores the data of the each step and its precedence rules, the functions and their parameters, the static and dynamic values of the parameters, the errors, etc.

The paper details the principles of the "Programming by steps", explains the reasons, which originally motivated the development of the method, and defines the principal requisites to build an application system. Future aspects of the implementation, as well as advantages/disadvantages of design, implementing and maintaining the system are stated.

The paper includes a comparative analysis between the "Programming by steps" and another two methods of software engineering: the "Rapid Prototyping" and the "Component-based Design". Integrative comments and conclusive remarks are provided in the conclusion of the paper.

1 New changes in the evolution of the software methods for analysis, design and programming

The fast evolution of technologies reflects a boom of software applications' requests. The result is a larger application domain, which is more diversified and/or specialised, that requires new software methods for analysis, design and programming in order to develop open applications, which may be transparent and easy-exportable from one domain to another.

Other characteristics of the actual requests are the speed, the on-line and real-time features, based on the Internet and telephony's services. In these cases, it is necessary to find out solutions to permit the adaptation and the improvement of the existing software without interrupting the system. Also, it is recommended to maintain the operative execution by trying to control and reduce the hardware costs.

*PhD Student of the Academy of Economic Studies, Bucharest, Romania, e-mail: oana.rs@tiscali.it

The growth of the number of the applications¹ and the increased complexity of the information systems may produce problems for the analysts and programmers. Therefore it is necessary to develop new methods that shall simplify the work for the programmers.

“Programming by steps” is a method that can be used in order to analyse and design software in the following cases:

- for systems characterised by many applications belonging to the same family with common functions, or that may become common after a standardisation process;
- for real-time systems, where changes or maintenance should not cause the interruption of the existing services;
- for real-time systems, where it is necessary to test the newer releases without interrupting or damaging the older ones.

“Programming by steps” offers some advantages:

- standardisation of the problems in the analysis process, reducing time and resource consuming;
- higher degree of flexibility and greater speed in the programming activities for the applications in the same domain;
- better visibility that helps the control and error correction activities.

The method is based on the old flow-charts in order to obtain, not only a logical representation of the problem to be solved, but also the real-time functioning of the software itself.

2 What is “Programming by steps”?

Supposing one should write several programs that utilise functions belonging to a defined group and which show some homogeneities.

The traditional solution is based on:

- grouping several functions into a library;
- projecting more specific applications which call for different single functions with defined parameters, each one to solve some problems.

This solution supposes that a programmer writes the code for these applications: one defines the new variables, chooses and picks out the necessary functions from the library, integrates them, and after tests the program’s functionality (see Fig. 1).

¹It is very difficult to imagine any activity that could not be usefully touched by the information technology, even if the effective informational process is not concluded yet.

In such a situation the difference between one application and the other is represented by the used functions' set and their priority. How could this work be done without writing a new program code every time? Is there an alternate solution?

The alternative is offered with the method "*Programming by steps*": a unique application accesses the functions' library through its own interface. The application consists of an "engine" that follows different logical flows memorised in either internal or external tables. One flow is used for every problem to be solved.

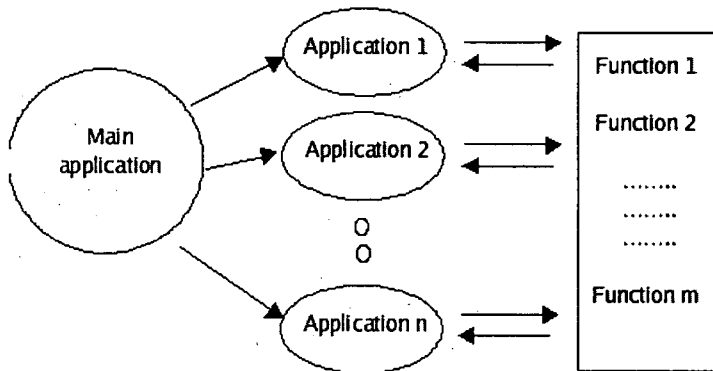


Figure 1: A main application launches many specific applications using the same functions' library

The analysis process of a new problem may be resumed in building up and implementing the logical flow in data tables. The execution "engine" pursues this logical flow and launches the various functions, building up the application by itself in a stepwise manner during run-time (see Fig. 2). Indeed it is necessary that the main application find all the functions inserted in the logical flow within the functions' library.

The design and implement phases supposed for every new application:

- to define the logical flow represented by steps to be done and their relative functions;
- to identify the values of all functions' parameters and the link/order among them;
- to put these elements together through their descriptive information inserted into a specific structure of tables (subsequently described);
- to launch the main application with a reference to the tables specific to certain applications.

The major advantage is that the main program is independent from the content of the logical flows. It knows only the modality to pursue such a flow, and to

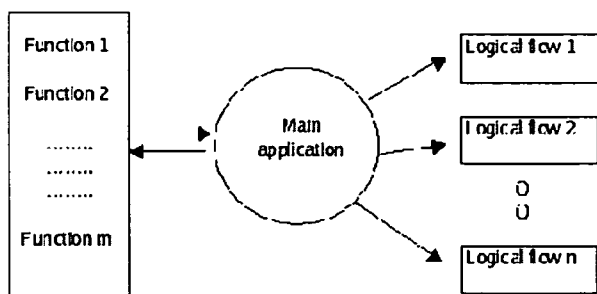


Figure 2: A main application is used for many specific logical flows, and it is linked to the functions' library for this goal

transfer different values from one step to another. Several concepts which are necessary to comprehensively understand the method "Programming by steps" are presented below.

2.1 Logical flows and flow-charts

In order to solve a specific problem it is necessary to understand its prerequisites and the different situations that could appear or influence it, and to intercept its possible results. It is seldomly possible to represent the solution through a *logical flow* of events or situations.

At the end of the 60's, Edsger Dijkstra and others proposed three logical constructions to represent the logical flow of a program. These are: "sequence, condition and repetition. The *sequence* implements the procedures' steps, which are essential for each algorithm. The *condition* provides the capacity to elaborate selectively, based on determined logical conditions; whereas the *repetition* consents to perform cycles. All these constructions are fundamental for structured programming, which is an important technique of projecting the components' level. In practice, every construction has a predictable logical structure with the entry at the superior part and the exit at the inferior part, which allows to easily follow the procedure's flow"².

"Each programme can be projected, independently for the application area or technical complexity, by using only three types of structured constructions"². Its *logical flow* provides a precise specification of the elaboration that means the events' sequence, the iterations, and the decisions' points, using certain data structures.

The *flow chart* is "a pictorial representation of the steps in a process, useful for investigating opportunities for improvement by gaining a detailed understanding of

²Unofficial translation from the Italian version of R. S. Pressman, "Principi di ingegneria del software", 2000, pp.442-443

how the process actually works”³.

“Flow charts have been used for so long that no one individual is specified as the *father of the flow chart*”⁴. “The flow chart is a means of communicating information. It must be able to communicate the steps in a process clearly and unambiguously”⁵.

The New Oxford Dictionary defines the *flow-chart* (flow diagram) as: “a diagram of sequence of movements or actions of people or things involved in a complex system or activity; [...] a graphical representation of a computer program in relation to its sequence of functions (as distinct from the data it processes)”⁶.

As mentioned above, for each initial application from Fig. 1 a set of information describing the own logical flow is presented: its steps and its functions, which have to be performed at every step.

2.2 Functions

The second part of the flow chart definition written in the New Oxford Dictionary introduces another important concept: *the function*. According to the same dictionary, the *function* is “a basic task of a computer, especially one that corresponds to a single instruction from the user”⁷, or, mathematically, “a relation or expression involving one or many variables”⁷. Usually, a function is a “black box” that returns a value⁸. The user interacts with the function through the function’s parameters⁹, and possibly through global variables.

In “Programming by steps” a *function* is a routine (application, relationship, or transformation) that accepts a certain number of input parameters, uses them for its elaboration and returns some results through its output parameters.

Let us consider a function f_j , having x_j input parameters and y_j output parameters:

$$\begin{aligned} f_j : I_j &\rightarrow O_j, & 1 \leq j \leq n & \quad j, n \in N & (1) \\ I_j &= I_1 x I_2 x \dots x I_h x \dots x I_{x_j} & 1 \leq h \leq x_j & \quad h, x_j \in N \\ O_j &= O_1 x O_2 x \dots x O_l x \dots x O_{y_j} & 1 \leq l \leq y_j & \quad l, y_j \in N \\ f_j &= f_j(i_1, i_2, \dots, i_h, \dots, i_{x_j}, o_1, o_2, \dots, o_l, \dots, o_{y_j}) \end{aligned}$$

³See ISO 9004-4 “Quality management and quality system Elements - Part 4: Guidelines for quality improvement”, 1993, p. 13. This standard has fixed a symbol set that ensures instant recognition by everyone in order to allow a unique representation of the fundamental constructions and logical flows.

⁴See J. R. Clauson, T. Glenn, J. A. O. Hunter, “Index of quality control tutorials”, 1995, p. 2

⁵See T. Burns, “A Fresh Look at Flow Charting”, p. 1

⁶See “The New Oxford Dictionary of English”, Clarendon Press, Oxford, 1998, p. 707

⁷See “The New Oxford Dictionary of English”, 1998, p. 743

⁸In programming a function may return a value, a vector or structure. The mathematical definition becomes larger. We know that a function may return a value or a memory’s address (a pointer), that would contain everything. The large concept of functions is covered by functions and procedures in some programming languages.

⁹The parameter may have different values (see note 10): number, string, date, vector, matrix, pointer, structure, image, etc.

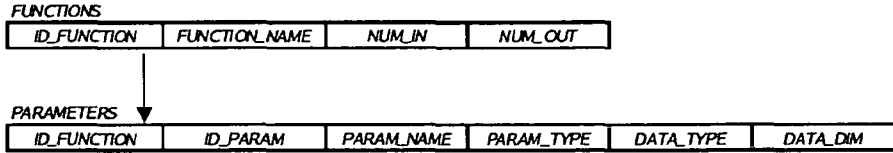


Figure 3: Functions and parameters

where

i_h – the input parameter, $i_h \in I_h$

o_l – the output parameter, $o_l \in O_l$

x_j – the number of input parameters for the function f_j

y_j – the number of output parameters for the function f_j

I_h, O_l – sets of values having homogeneous types of data (see the last footnotes)

Let us consider a set of functions defined above:

$$F = \{f_j\} \quad 1 \leq j \leq n \quad j, n \in N \quad (2)$$

where

n – the number of functions in the library

Each parameter, i_h or o_l , is characterised by its type and dimension. In “Programming by steps” the parameters are like doors, which permit the entrance and exit of values in the function’s body. Let us consider the generic parameter p_i :

$$p_i : PT * DT * N \rightarrow P_i \quad 1 \leq i \leq x_j + y_j \quad i, x_j, y_j \in N \quad (3)$$

$PT = \{\text{Input, Output}\}$

$DT = \{\text{Boolean, short, long, string, pointer, etc.}\}$

$P_i = I_h \text{ or } O_l$

$p_i = p_i(t_i, dt_i, d_i(dt_i))$

where

t_i – the type of parameter p_i : input/output

dt_i – the data type contained by the parameter (Boolean, short, long, string, pointer, etc.)

$d_i(dt_i)$ – the dimension of the parameter (that may be dependent of the data type), $d_i(dt_i) \in N$

PT – the set of parameters’ types

DT – the set of data types that depends on the programming language

One function may be a simple function or a macro-function that performs a certain group of activities. For example, it could only print a value on the output terminal or play a welcome message on an answering machine.

The functions are integrated in a library and called by a main application (see section 2.4). The information that is related to the functions’ definition is represented in a database system in this modality:

The first table, FUNCTIONS, stores information about the functions:

- the function's identifier;
- the function's name;
- the number of input/output parameters (x_j, y_j) .

The second one, PARAMETERS, contains information about the parameters of every function: the function's identifier, the parameter's identifier, its name¹⁰ and type (input/output), the accepted types¹¹ and the maximum size of respective types¹².

2.3 Steps

Returning to the structured programming, the body of a program (between start and stop) is composed by a sequence of steps.

"Programming by steps" considers a *step* as the smallest logical part of a program, which can be associated with a function to be executed. The decomposition's level of a logical flow in its steps is inversely proportional to the standardisation's degree of the functions: the more abstract the functions, the fewer are the steps.

Consider S the set of steps from a logical flow:

$$S = \{s_i\} \quad 1 \leq i \leq m \quad i, m \in N \quad (4)$$

where

s_i – the step

m – the number of steps to be done

Each step s_i is described by the following information:

$$s_i = s_i(t_i, \{(c_{ik}, s_{n_k})\}_{1 \leq k \leq n_i}, f_i) \quad 1 \leq i, n_k \leq m \quad (5)$$

$$s_i \in S, \quad t_i \in T, \quad f_i \in F, \quad c_{ik} \in C, \quad s_{n_k} \in S \quad i, k, m, n_k, n_i \in N$$

where

t_i – the type of step s_i

f_i – the function associated with the step s_i

¹⁰A generic name is used in order to identify the parameter.

¹¹The types and dimensions depend on the used programming language and/or database management system. In this example SQL Server 2000 and C are considered. The information is only descriptive, and it should be used during the data input process or the program's execution. We note that using a varchar data type, it is possible to memorise different data in the database, and after to utilise them independently of their types, because of the implicit conversions done by the SQL Server 2000.

¹²It is recommended to use a type compliant with a major number of possible received values and to perform internal conversions either at the function calling level (main application) or at the function execution level (library).

(c_{ik}, s_{n_k}) – one of the couples (*condition, successive step*) which determines the behaviour of the logic flow: if the condition c_{ik} is fulfilled, the step s_i will be followed by the step s_{n_k} , and $1 \leq k \leq n_i$, $1 \leq n_k \leq m$ ¹³.

n_i – the number of conditions (situations) that may appear

S – the set of steps

F – the set of functions

C_i, T – the sets of conditions and types defined below

The “Programming by steps” proposes the set of the steps’ types below:

$$T = \{I = \text{Iteration}, D = \text{Decision}, S = \text{Jump}, L = \text{Loop}\} \quad (6)$$

Let’s see the meaning of each type in relation with the fundamental constructions proposed by the structured programming: sequence, condition and repetition (see section 2.1).

The I type (*iteration*) corresponds to *sequence* construction. If we consider the actual step s_i , the step s_{i+1} will succeed it. There is no condition to be evaluated, and the step number will increase with a fixed iteration, equal to 1 (see Fig. 4a).

The D type (*decision*) corresponds to *decision* construction. The number of conditions to be evaluated is unlimited (like in a multiple selection) and the next step will be calculated based on the condition’s evaluation. Let us consider C_i the set of conditions, which are supposed to cover all possibilities, which can arise in the decisional step s_i :

$$C_i = \{c_{ik}\} \quad 1 \leq i \leq m, \quad 1 \leq k \leq n_i \quad i, k, m, n_i \in N \quad (7)$$

where:

c_{ik} – the condition associated with the step s_i

n_i – the number of possible conditions for the step s_i

m – the number of steps in the flow

If the condition c_{i1} is true, then the step s_{i+1} will follow the step s_i . If the condition c_{i2} is true, then the step s_{i+2} will follow the step s_i , and so on, until the last condition (if the condition c_{in_i} is true, then s_i will be succeeded by s_{i+n_i})¹⁴. All these branches will meet in the next step s_{i+n_i+1} . The conditions are discrete and finite (see Fig. 4b)¹⁵.

¹³For simplifying, we will consider the relation between the step’s index n_k and the condition’s index k in the couple (condition, successive step) as linear: $n_k = i + k$. Note that “Programming by steps” is not limited to this linearity.

¹⁴Remember that when running an application one condition will only be true at a certain moment of time. The flow will pass through a unique branch.

¹⁵The problem is to transform a set of continual conditions in a set of discrete conditions. For example, if the condition is: “if $x > 0$ then true else false”, then a function will be used in order to translate it like here: $f(x) = (\text{unknown char})$ if $x > 0$ return 1, else return 0. The condition becomes: “if $f(x)=1$ then true, if $f(x)=0$ then false”. We have obtained a discrete condition from a continuous one.

¹⁸It is greater than 1 for a D step, otherwise it is equal to 1.

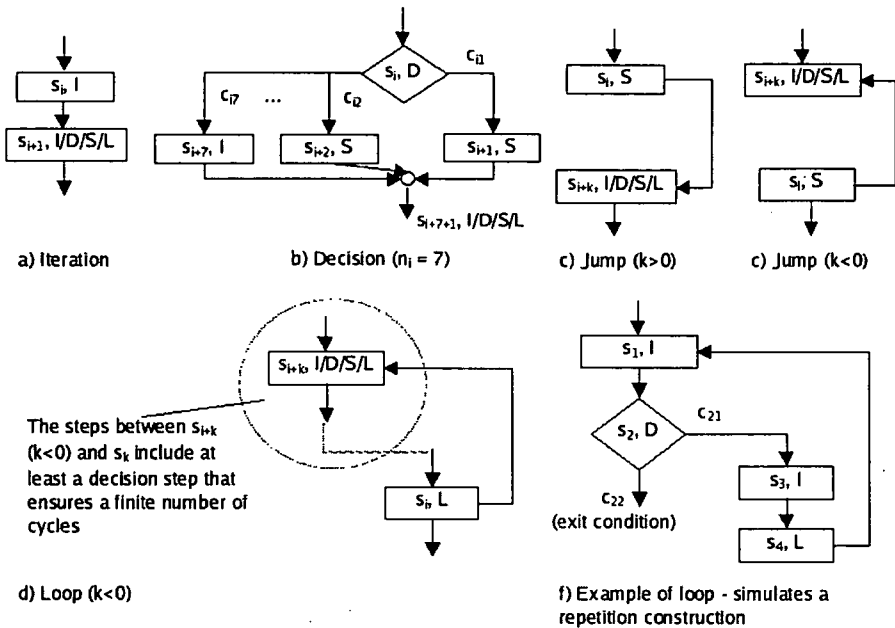


Figure 4: Example of steps' types defined in the "Programming by steps" method

The table NEXT_STEPS is related to the couples (condition, successive step) that force the execution: the current step's identifier, the condition's identifier, the value for which the condition is true and the corresponding next step to be executed (when that condition is realised). For the types which differ from decision, the condition can not be estimated (ID-COND=0, VAL-COND=<NULL>). DEF_STEPS contains the set of types, with their description. The main application has implemented a special mechanism in order to automatically treat the different types of steps (see section 3 of this paper).

We can observe that not all the information of the logical flow is memorised, because there is no exchange of data between the functions from one step to another, thus we haven't obtained a functional system yet.

The possible parameters' values have the following characteristics:

- they may be fixed or variable, deterministic or non-deterministic;
- they may be dependent or independent of the step in that are executed, as well as the precedent steps;
- they have a certain type of data and a specific dimension.

To solve the data transfer, "Programming by steps" introduces a new concept: a *field* associated with one or many parameters, which is a part of a specific table:

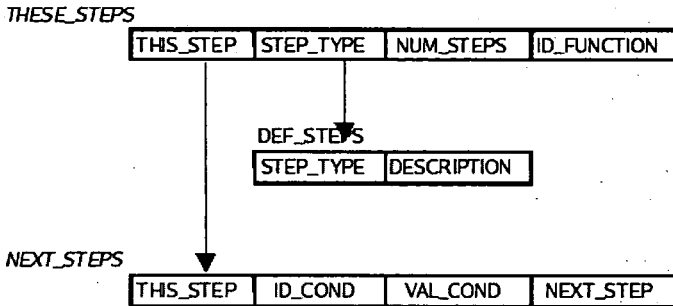


Figure 5: The general structure of the tables with information for the flow building

TRANSFER_VALUES. Transferring data from one step to another means associating the same field with two parameters: one output parameter appertaining to the first function and one input parameter belonging to the successive function (indicated by the specific logical flow). The first parameter will scatter the output value in the field mentioned above, and the second one will gather it to use in its function¹⁹.

For example, we can choose to associate the o_{it} output parameter of the f_i function with the `<FIELD_1>` field at the s_i step. After we may associate the i_{jk} input parameter of the f_j function with the same field in order to receive the contained data at the s_j step.

The link between the parameters and their fields will be done according to the step where the function is called (see **FIELDS.THIS_STEP** table). This table contains the step's identifier, the parameter's identifier and the field's identifier (associated with the field's description in **FIELDS** table).

The **TRANSFER_VALUES** table is composed of fields having names, types, and initial values as described in **FIELDS** table. If many applications use the same flow, many rows will be inserted in the table **TRANSFER_VALUES**, and each application will identify its row by a number (record number). The value memorised into `ID_FIELD` field (see tables **FIELDS**, **FIELDS.THIS_STEP** and **FIELDS.NEXT_STEP**) indicates the position of the corresponding field in the **TRANSFER_VALUES** table's structure. Please observe that we may associate now the field `<FIELD_1>` with other inputs parameter for reusing the value memorised there, or with other output parameters for loading it with a new value.

Let's consider another situation. Supposing the same function is used in the same flow twice, with different values each time. There are two solutions:

- to associate new fields with the respective parameters, memorising their initial

¹⁹Do you remember the interpretation of the parameters? Let's imagine now the field like a room between two doors (the parameters): the value goes out from the first function and enters in this room using one door. When it is necessary, the other door is open and the value leaves the room, entering in the successive function.

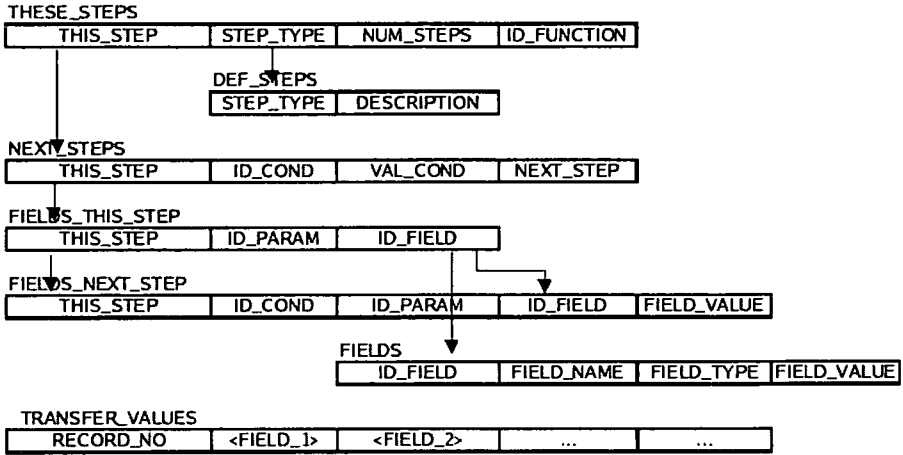


Figure 6: A logical flow implementation in a database management system

values into the FIELD-VALUE fields (see FIELDS table);

- to use the same associated fields and to reset the values.

In the second alternative another data set will be used, that specifies the initialisations to be made for a successive step in a known situation, given by the couple (condition, successive step): FIELDS_NEXT_STEP. The parameters, the fields and the associated values are memorised here, and will be set for a certain condition that is realised during the application's execution. The table contains the step's identifier, the condition's identifier (in order to know the branch), the parameter's identifier, the field's identifier and its initial value.

The "field" concept offers a great flexibility in the functions' management. As described above, the fields can be initialised with static values at the beginning or at a certain moment during the run-time. They also permit the data transfer between steps and functions.

Our database schema may be completed like in Fig. 6.

Resuming, we can consider the logical flow as an oriented graph. In each node a certain function is realised and the next step is decided. Parameters' values may be transferred from one node to another, using FIELDS_THIS_STEP table, and some parameters may sometimes be re-initialised, using FIELDS_NEXT_STEP table.

The information related to the step execution may be represented as:

$$s_i = s_i(t_i, \{(c_{i1}, s_{i+1}), \dots, (c_{ik}, s_{i+k}), \dots, (c_{in_i}, s_{i+n_i})\}, f_i(v_{i1}, v_{i2}, \dots, v_{ih}, \dots, v_{iz}, w_{i1}, \dots, w_{il}, \dots, w_{iy_i})) \quad (8)$$

$$\begin{array}{llll}
s_i \in S, & t_i \in T, & f_i \in F, & c_{ik} \in C, \quad s_{i+k} \in S \\
1 \leq i, i+k \leq m, & 1 \leq k \leq n_i & & i, k, m, n_i \in N \\
1 \leq h \leq x_i, & 1 \leq l \leq y_i & & h, l, x_i, y_i \in N
\end{array}$$

where

s_i – the step s_i

t_i – the type of step s_i

f_i – the function specific to the step s_i

v_{ih} – the value s_i of the input parameter i_h specific to the step s_i

w_{il} – the value of the output parameter o_l specific to the step s_i

(c_{ik}, s_{i+k}) – one of the couples (condition, successive step), for step s_i , where $1 \leq k \leq n_i, 1 \leq i+k \leq m$

2.4 The Main Application

The main application will be built in order to roll over the logical flow memorised in the tables, following the procedure presented in Fig. 7.

In this representation, an iterative alternative was chosen to operate the whole logical flow. The idea to build such a program comes from the backtracking engine used in the nonprocedural language Prolog, that can be stopped only with a specific instruction (in our case: the value of the next step is equal to 0)²⁰.

In the case of “Programming by steps” method, the application that follows the flow’s evolution consists in a “do-while” cycle that operates until it receives a specific exit instruction. A flow step is performed at any iteration, so that a specific function is launched, then the next step is identified. The zero value for the step represents the exit from the logical flow (and also from the program).

In this way the logical flows can be built physically, the main program has an role of execution, but is transparent for the content of execution. If the functions have been analysed and projected to be usable in many and different situations and programs, then the programmers work is reduced only to organise the functions in the necessary order and to fill in some linking information in the database tables.

From the Fig. 7 we may deduce the mechanism on which the method “Programming by steps” is based and from where its name comes from: it loads one step by one and executes each associated function. The sequence of steps is linked to the specific characteristics for the problem and for the domain.

3 How to apply “Programming by steps”?

“Programming by steps” method requires the next stages (see Fig. 8):

1. Domain Analysis
2. Main application development

²⁰It may be used a recursive method that will be more suggestive when the logical flow is interpreted like an oriented graph.

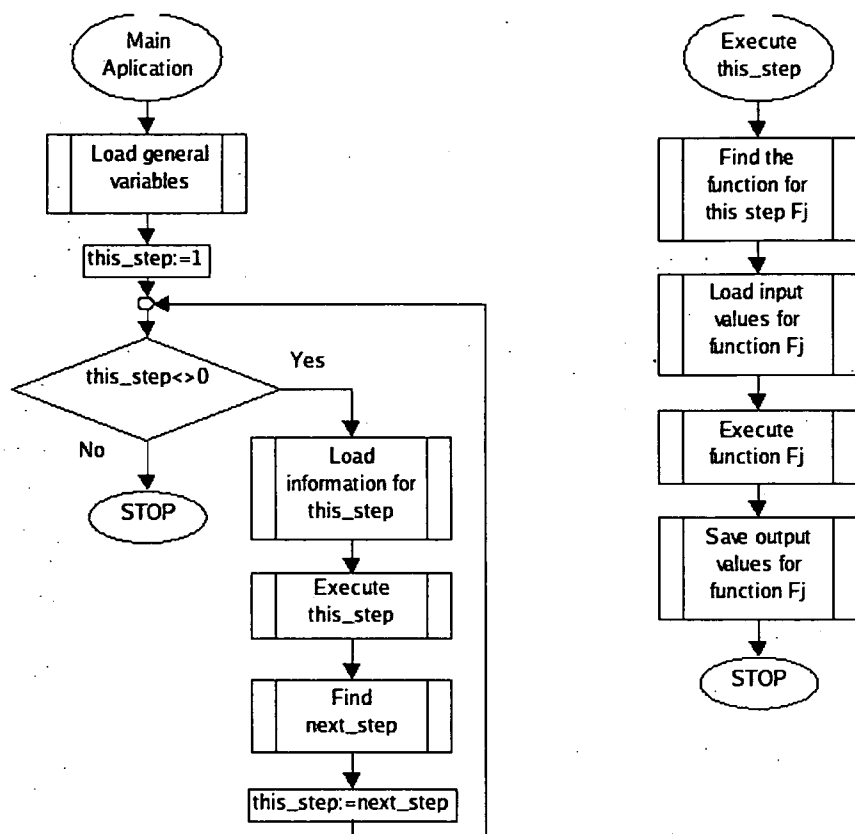


Figure 7: The behaviour of the main application that executes the logical flow

3. Functions development
4. Functions integration in the Common Library
5. Specific problem analysis; if there are new functions to be developed go to 4, else go to 6
6. Flow chart design and implementation for the problem solution
7. Application testing and homologation

For all these stages a short description will be presented²¹.

1) Domain analysis

²¹Note: More homogeneous functionality has the analysed domain, more simply is to apply the method. If the analysis is correct and the functions present a high degree of standardisation, the steps 1,2,3, and 4 are rarely used.

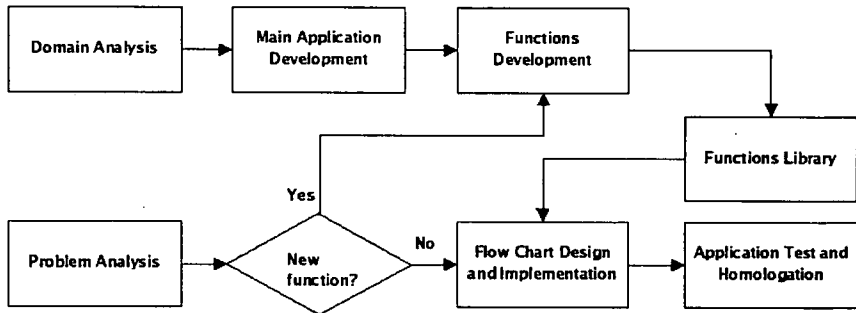


Figure 8: The process model for the method “Programming by steps”

The general description of the domain analysis can be taken over from the oriented objects design method, with the following definition: “the domain analysis for the software consists of the location, analysis and specification of the common requirements in a specific application sector in order to reuse some parts of them in many projects from that sector”²².

In the “Programming by steps” method, the domain analysis looks for identification of general characteristics of the sector, the location of already existing applications, the identification of future and present requirements, the identification of main requested functions, and the setting up of reusing projecting standards.

2) Main application development

This phase permits to build the main application with its interface towards the function library (internally or externally implemented).

The main software reads the information of each step, gathers the input values of the corresponding function and interprets them, runs the function and scatters the output results in associated fields, manages the checking of all the steps, and ensures the execution of the entire system well (see Fig. 7).

The main application’s characteristics are defined in the way of interacting with the functions on one side, and with the data on the other side. The modality to execute the steps (recursive or iterative) is chosen, and the engine that follows the logical flow is projected.

In this stage the database structure is implemented, which stores the data of the steps and its priorities, functions and its parameters, static and dynamic values of the parameters, errors etc. The structures that implement these functions are projected.

²²Unofficial translation from the Italian version of R. S. Pressman in “Principi di ingegneria del software”, 2000, pp.598, that refers D. G. Firesmith with “Object Oriented Requirements Analysis and Logical Design”, 1993

3) *Functions development*

The analysis starts from the functions identified at the first stage. The input and output parameters are defined in order to be possible to use a function in as many situations as possible.

The name, the data type and its dimension are established for each input/output parameter. In order to raise the usage of a function it is recommended that each parameter accept many data types. Some conversion mechanisms will be applied inside each function.

The data structures defined in second stage and represented in Fig. 3 are loaded. The functions can be written in a high level language, or in the language incorporated in the chosen database management system.

4) *Functions integration in the Common Library*

The role of this stage is to integrate the functions, which were projected at the third stage, into a library. A friendly interface between the main application and the library shall be created in order to reuse the functions if necessary. However, the application will interpret the information memorised at the second stage to access and run the functions.

5) *Specific problem analysis*

In this phase the problem must be identified within the area. It is necessary to find similar problems and their possible solutions. If there is more than one solution, one must be chosen and its corresponding functions must be defined. When all of these functions are defined, we can go directly to the sixth stage.

If the analysis reflects the necessity of new functions, we will return to the third stage – *Functions development*, until all the new functions are integrated into the Common Library.

6) *Flow chart design and implementation for the problem solution*

The projecting and implementation of the logical flow represent the effective programming process. This stage's goal is the structure loading with the specific values for the problem's solution.

It is based on the chosen solution at the fifth point, where the necessary functions were identified and defined, and it continues with the establishing of functions' succession. The logical flow is built up asking questions like: "What really happens next in the process?", "Does a decision need to be made before the next step?", or "What approvals are required before moving on to the next step?". The data structures from Fig. 6 are loaded. The initial values are established for each step.

7) *Application testing and homologation*

After putting all the pieces together, the logical flow execution must be tested by the main application. With the flow chart already built, we need to make

a test plan that covers all the different paths and to execute it, correcting the eventual differences from the thought-established solution.

4 Comparative analysis between the “Programming by steps”, the “Rapid Prototyping” and the “Component-based Design”

This section contains a short presentation of two other design method (Rapid Prototyping and the Component-based Design) and some comparative characteristics between these methods and the “Programming by steps” method.

The “Rapid Prototyping” method

“The prototyping paradigm may be <closed> or <open>”²³. In the first case the prototype is considered a “quick and dirty” affair, used like a communications aid between users and developers. Once the sought information has been obtained, it is discarded and conventional software design ensues (see Fig. 9). In the second case, the prototype becomes the central focus of the process model, called “evolutionary prototyping”. The prototype is scoped, scheduled, resources are allocated and refined as depicted in Fig. 10²⁴.

“One solution for the “Rapid Prototyping” consists of the prototype assembling (instead of building it), using the existing software components. An existing software product can be used like a prototype for a new product. In a same sense, this is a reuse form applied on the prototyping”²⁵.

In the case of the “Programming by steps” method, one or many existing applications may constitute a prototype model for a new one. In this way a new problem will be quickly understood and resolved with predictable results. The old functions are all homologated and their use will be secure and free of errors. In case of detected errors, the corrections automatically touch the old applications (logical flows) which therefore makes the maintenance process easier. But the model for a new application is neither a closed prototype, nor an opened one, because it is more, it is a functional model, with correct results and acceptance by the client.

The “Component-based Design” method

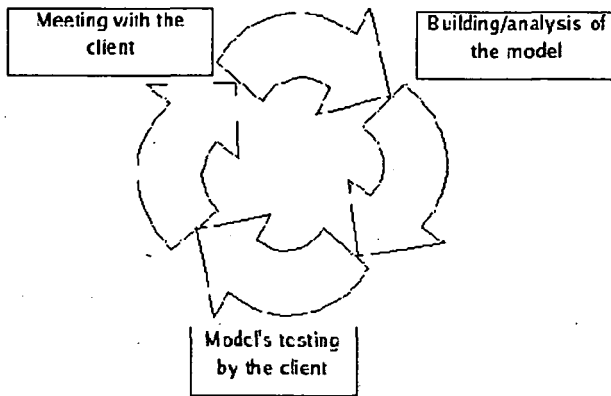
“Component based development offers a vision of plug and play software development”²⁶. “The Component-Based Software Engineering process is drastically different from the conventional software development process: it’s integration-centric

²³Unofficial translation from the Italian version of R. S. Pressman, “Principi di ingegneria del software”, 2000, p. 301

²⁴See R. L. Vienneau, R. Senn – “A state of the ART Report: software design methods”, 1995, p. 12

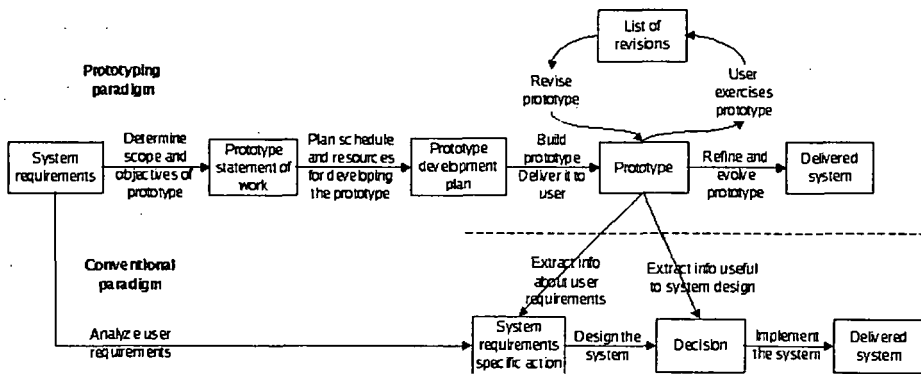
²⁵Unofficial translation from the Italian version of R. S. Pressman, “Principi di ingegneria del software”, 2000, p. 302

²⁶See M. Collins-Cope, D. Deveaux, P. Frison, H. Matthews, G. Pour, “Component Based Development: Software Architecture, Component Models and Teaching”, p. 1



Source: Unofficial translation from the Italian version of R. S. Pressman, "Principi di ingegneria del software", 2000, p. 33

Figure 9: The prototype paradigm



Source: W.W. Agresti, "What are the New Paradigms?"

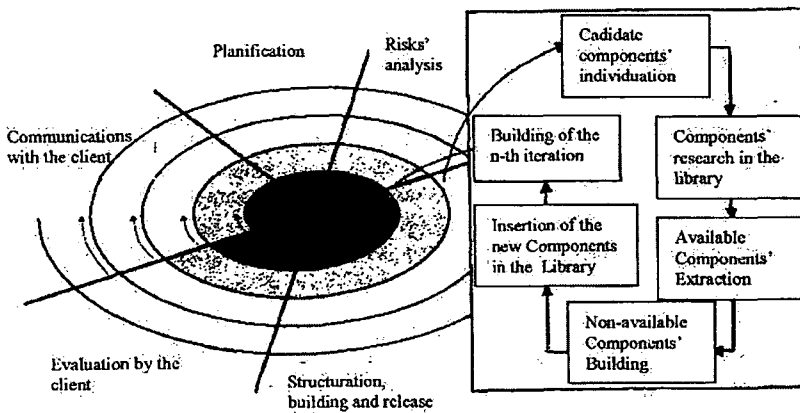
Figure 10: The Prototyping Paradigm and its Relationship to the Conventional Software Development.

as opposed to development-centric. This is a real challenge for developers, but also for teachers. New skills should be emphasised:

- connections between analysis, design and programming,
- documentation use and its construction,

- testing and validation technologies,
- reliability and trustability,
- software project management”²⁷.

“When building procedural code, common behaviour is extracted into some kind of function or subroutine that can be reused in some way, either by having all users call the same function implementation, or by copying the function into code at compile time.



Source: Unofficial translation from the Italian version of R. S. Pressman, "Principi di ingegneria del software", 2000, p. 45

Figure 11: Component-based Development

When designing a component-based solution, it is possible to extract a common behaviour so that multiple consumers can use it”²⁸.

The “Programming by steps” method may be situated between these two approaches: the common behaviour is extracted into functions like in a procedural method. But, the way in which they are used is nearer to the component-based method, because the main application calls them like external components. The logical flow is composed piece by piece for every solution, reusing the functions. The functions may be used to solve several problems of a specific domain, as well as in a different application domain, if they have the requested functionality and a recognisable interface²⁹.

²⁷See M. Collins-Cope, D. Deveaux, P. Frison, H. Matthews, G. Pour, “Component Based Development: Software Architecture, Component Models and Teaching”, p. 1

²⁸See K. McInnis, “Component-based Design and Reuse”, 1999, p. 2

²⁹The problem of CPU's usage shall be solved using a multiprocessor computer.

Table 1: Comparative requirements to apply the design methods: Rapid Prototyping, Component Based Development (CBD) and "Programming by steps"

Requirements	Closed Prototyping	Open Prototyping	CBD	Programming by steps
Application domain well known	*	*	*	*
Problems may be modelled	*	*	*	*
Accurate and stable requirements	-	*	*	*
Ambiguous and contradictory requirements	*	-	-	-
Possibility of reuse	-	*	*	*

Source: adapted from R. S. Pressman, "Principi di ingegneria del software", 2000, p. 302

"The component based development model represented in Fig. 11 incorporates several characteristics of the spiral model. It presents an evolutive nature³⁰ and requires a software development iterative approach. However this development model creates applications starting from software components ready to be used (the classes)"³¹.

We see that also the "Programming by steps" method has an evolutive nature. One could remark on the similarity between the process of component individuation and design on one hand, and the process of function identification and design, on the other hand. One function is like a component, with its functionality and its interface implemented into the database tables (see Fig. 3).

In the next table we may see some requirements that could be used to choose one of the methods:

Different advantages and disadvantages of the three methods are listed in the Table 2.

5 Conclusions

The use of a high level language with a good database management system adds some advantages to the "Programming by steps" method:

- the velocity and accuracy of computing offered by the programming language;

³⁰See R. S. Pressman in "Principi di ingegneria del software", 2000, p. 44, that refers the article of Nierstrasz, O., Gibbs, S., Tsichritzis, D., "Component-Oriented Software Development", 1992, pp. 160-165

³¹Unofficial translation from the Italian version of R. S. Pressman, "Principi di ingegneria del software", 2000, p. 44

Table 2: Comparative advantages and disadvantages

Advantages and disadvantages	Rapid Prototyping	CBD	Programming by steps
Rapid understanding of the requirements	*	-	-
Rapid application building	-	*	*
Reuse possibilities	*	*	*
Reducing of the development cycle	-	*	*
Reducing of the project costs	*	*	*
Increasing of the productivity	-	*	*
Possibility to use wizards	*	*	*
Possibility to use non-expert personnel ³²	-	-	*
Prototype may be shallow and narrow	*	-	-
Components too complicated	-	*	-
Integration issues between main application and database implementation	-	-	*

- faster and secure data access and manipulation, without concurrency problems offered by the database management system;
- larger volume of data that could be stored inside of the same database management system.

The idea of the functions' incorporation inside the database management system, and the ability to be updated for different problems without touching the source in the high-level language, raises the flexibility of the program and offers new development's perspectives.

"Programming by steps" is a new method that designs and implements a mechanism that executes specific steps corresponding to a logical flow and permits to build new applications only configuring some tables with the steps and the functions necessary in that case. A new program may be designed building the succession of steps with the necessary functions for the respective situation. In other words, the software engineer would fill in some fields in the tables with some values in order to complete the logical flow and then to run the program.

In a future release a non-specialist user could also be the "writer" of the software. One possible solution is to implement a system, with an adapted graphical interface (for example, an icon for each function). The user will have to choose the icons and to connect them together in a logical flow. Another solution is to use a flow-charting software for drawing the logical flow and then automatically to convert it in the requested information in order to fill in the specific tables.

The "Programming by steps" method is based on the reusable functions that may be designed without changing the main application. That offers flexibility and manageability in obtaining new software releases.

³²This feature refers to the situation when the library has already been implemented.

References

- [1] Agresti, W. W. "What are the New Paradigms?" In Agresti, W.W. (ed.) "New Paradigms for Software Development", Washington, DC: IEEE Computer Society, 1986
- [2] Burns, T. "A Fresh Look at Flow Charting",
<http://www.q-skills.com/flowchrt.html>
- [3] Chichernea, V., Botezatu, C., Iacob, I., Fabian, C., Mihalcea, R., Goron, S. "Proiectarea sistemelor informatice", Sylvi, Bucharest, 2001
- [4] Clauson, J. R., Glenn, T., Hunter, J. A. H. "Index of quality control tutorials", Clemson CQI Server Copyright (c) 1995 by Clemson University Portions Copyright (c), 1995
<http://deming.eng.clemson.edu/pub/tutorials/qctools/flowm.htm>
- [5] Collins-Cope, M., Deveaux, D., Frison, P., Matthews, H., Pour, G. "Component Based Development: Software Architecture, Component Models and Teaching", CBD-TOOL, 2000
- [6] Firesmith, D.G. "Object Oriented Requirements Analysis and Logical Design", Wiley, 1993
- [7] McInnis, K. "Component-based Design and Reuse", Castek, 1999
- [8] Mihalca, R., Tataru, A. "Realizarea produselor program. Metode si tehnici de analiza si proiectare structurata", Scripta, Bucharest, 1994
- [9] Nierstrasz, O., Gibbs, S., Tsichritzis, D. "Component-Oriented Software Development", CACM, vol. 35, no. 9, September 1992, pp. 160-165
- [10] Pressman, R. S. "Principi di ingegneria del software", third edition, McGraw Hill, Milan, 2000
- [11] Vienneau, R. L., Senn, R. "A state of the ART Report: software design methods", ITT System Corporation, Griffiss Bussiness and Technology Park, Rome, 1995 - <http://www.dacs.dtic.mil/techs/design/Design.toc.html>
- [12] *** "The New Oxford Dictionary of English", Clarendon Press, Oxford, 1998
- [13] *** ISO 9004-4 "Quality management and quality system elements, Part 4: Guidelines for quality improvement", Geneva, first edition, 1993

Two Content Protection Schemes for Digital Items

Paula Steinby*

Abstract

Modern techniques make digital articles easy to copy and manipulate. *Content protection systems* aim at protecting the rights of producers and distributors. These mostly rely on data encryption, digital watermarking, and special-purpose devices. In this paper, we describe two content protection schemes, both of which make use of tamper-resistant devices and device-dependent decryption keys. One of the schemes uses a modified El Gamal system, in the other one we combine watermarking with encryption.

1 Introduction

Consider a scheme where a digital article is distributed over an insecure channel. During the transmission, the data may be subjected to eavesdropping and transformations. The combination of digital data and modern techniques to handle it brings along some controversial possibilities. Producing identical or manipulated copies of a digitized item is easy, and devices and programs for this purpose are commonly available.

For the parties using the transmission channel, there is a need for *privacy* and *content authentication* (i.e. capability to detect any data manipulation). Owner of an item may require *copyright protection* and further, a possibility for *traitor tracing*, maybe even for *copy* and/or *use control* of the item.

Content protection systems have been designed to protect media producers and distributors. The existing tools are limited: data encryption, digital watermarking, tamper-resistant and special-purpose devices. Encryption contributes to the privacy of the parties as well as makes the data useless for those without means to decrypt it. Watermarking enables one to recognize the copyright owner, or even distinguish between each copy of the data. A unique label in every copy provides for traitor tracing. This type of watermarking is usually referred to as *fingerprinting*. Cryptographic protection gives privacy for the transmission, but there lies a fundamental weakness in it. Namely, one must remove the encryption at some point to reproduce the item, thus leaving the item without any protection whatsoever.

*Turku Centre for Computer Science, 20014 Turun yliopisto, Finland, email: pauste@utu.fi

Special-purpose devices may offer a solution to this problem. These are devices with some special features, designed in view of a certain operating system. In this work, we describe two content protection schemes, which both make use of special-purpose devices, and device-dependent decryption keys. In Chapter 2 we describe a scheme with a modified El Gamal system, where the device can recognize if the input is supposed to be given in encrypted form, and refuses to process such data if given in plaintext form. In Chapter 3 we sketch a scheme to combine encryption, watermarking and compression of the data. In both schemes, we assume the device to have a secret key which is not known to anybody outside the device.

Digital watermarking is the so far best technique to protect an item after decryption. As the ultimate goal of content protection (i.e. making producing illegal copies impossible) remains unachievable, digital watermarking introduces methods to make producing, distributing and using illegal copies of some data unattractive: difficult, risky or unprofitable. We use digital watermarking for both schemes discussed in this paper, in order to bring security even in the case where the security provided by encryption and/or the device failed.

2 A scheme with modified El Gamal system

In this section, we sketch a method to protect copyrighted digital items using techniques based on public-key cryptography and a tamper-resistant device \mathcal{D} . The items to be protected can be visual, aural, etc. We assume that a public-key interface is used: each \mathcal{D} is equipped with a public-key pair $(t_{\mathcal{D}}, s_{\mathcal{D}})$. The private key $s_{\mathcal{D}}$ is unknown even to the owner of \mathcal{D} . (It is a common practice that private keys are generated within smart cards such that no other unit will ever learn them.)

Our scheme has the following features:

- The data is delivered to buyers in encrypted form. The encryption is the same for all buyers. This is convenient, because then it is enough for the merchant to perform a single encryption on the data, and then make it freely available. We denote the (symmetric) encryption/decryption key by K .
- A tamper-resistant special-purpose device \mathcal{D} is needed to reproduce the item. The respective device-dependent key is needed for \mathcal{D} to be able to compute the decryption key K . Hence, this key is different for all buyers.
- Even if K was revealed, legal unhacked devices could not exploit it.

Consider giving up the last feature in the list. Then the key delivery could be realized by encrypting the decryption key K with the public device key $t_{\mathcal{D}}$, and sending it to \mathcal{D} . But if the decryption key was revealed, then unauthorized device-dependent keys could be easily computed, and thus legal devices would be compatible with hacked documents. Preventing this seems highly desirable.

In the following, we present the notation and the cryptographic primitives that will be used in our protocol.

- The tamper-resistant devices use an El Gamal type public key encryption system. The domain parameters common for all are p and g , where p is a large prime (1024 bits), and g is a generator modulo p . The private key of \mathcal{D} is $s_{\mathcal{D}}$ ($0 < s_{\mathcal{D}} < p - 1$), and the corresponding public key is $t_{\mathcal{D}} = g^{s_{\mathcal{D}}} \pmod{p}$.
- M (the Merchant) has an item for sale. We denote the digital representation of the item by I . Prior to delivery, I is encrypted using a symmetric encryption function E with key K . We denote $enc(I) = E(I, K)$.
- B (the Buyer) wants to buy the item, and he has the device \mathcal{D} with the key pair $(s_{\mathcal{D}}, t_{\mathcal{D}})$ to reproduce it from I .
- h is a cryptographic hash function with output length equal to the key length $|K|$.

For encryption, M could use some fast stream cipher. If, say, RC4 with key length 160 was used, then SHA-1 can be chosen for h with a 160 bit output.

The Protocol

The protocol proceeds as follows. Step 0, where M encrypts her data, is preliminary. Steps 1 and 2 constitute the purchase phase, and in step 3 B 's device \mathcal{D} decrypts and reproduces the data.

0. M selects a random $\alpha \in [1, p - 1]$ and computes $K = h(g^{\alpha})$. Then M computes $enc(I) = E(I, K)$, which is the data set for delivery.
1. B sends M a request for I together with his device public key $t_{\mathcal{D}}$.
2. M computes $r = t_{\mathcal{D}}^{\alpha} \pmod{p}$ and sends B the pair $\beta = (x, y)$, where $x = g^{\alpha} \cdot t_{\mathcal{D}}^r \pmod{p}$ and $y = g^r \pmod{p}$, together with $enc(I)$.
3. B inputs (x, y) and $enc(I)$ to his device \mathcal{D} . \mathcal{D} computes $K' = x \cdot (y^s)^{-1} \pmod{p}$, $r' = K'^s \pmod{p}$. Then it checks whether $y = g^{r'} \pmod{p}$. If this is the case, then \mathcal{D} computes $K = h(K')$ and $I = D(enc(I), K)$ and reproduces I .

The protocol is a modification of El Gamal system. The difference is that instead of picking a random r we choose $r = t_{\mathcal{D}}^{\alpha}$, thus tying the value to the device \mathcal{D} through its public key $t_{\mathcal{D}}$. If the protocol is properly performed, then \mathcal{D} obtains the correct key K in step 3. This is verified by observing that

$$K' = x \cdot (y^s)^{-1} = (g^{\alpha} \cdot t_{\mathcal{D}}^r) \cdot g^{-rs} = g^{\alpha} \cdot g^{rs} g^{-rs} = g^{\alpha} \pmod{p},$$

and hence $K = h(K')$. It follows that

$$r' = K'^s = g^{\alpha s} = t_{\mathcal{D}}^{\alpha} = r,$$

and hence the check in step 3 is always successful if r is of the right form.

Let us weigh a legal buyers chances to determine K after the purchase (without hacking \mathcal{D}). B knows the public key $t_{\mathcal{D}}$, and a pair (x, y) where $x = g^{\alpha} \cdot t_{\mathcal{D}}^r \pmod{p}$ and $y = g^r \pmod{p}$. Clearly, finding K is equivalent to finding g^{α} . Thus the task would be to compute $t_{\mathcal{D}}^r = g^{rs}$, given $t_{\mathcal{D}} = g^s$ and $y = g^r$. This is the famous Diffie-Hellman problem (DHP), for which no efficient algorithm is known. DHP is believed to be equivalent to the discrete logarithm problem (the equivalence has a partial proof, see [1]). The fact that r is of special form, $r = g^{\alpha s}$, does not seem to help, but one could as well select $r = \text{hash}(g^{\alpha s})$ to be on the safe side. Then also the check in step 3 changes to $y = g^{\text{hash}(r')} \pmod{p}$.

To be able to produce any device-dependent keys needed to decrypt and reproduce $\text{enc}(I)$, one must be able to compute r for a given t . For this purpose, one must know either α , or the respective s , since $r = t^{\alpha} = (x(y^s)^{-1})^s$. Note that the problem does not become any easier even if a hacker has discovered the encryption key $K = g^{\alpha}$; there is still the discrete logarithm problem to be solved for α . As we assumed that the knowledge of s is not available outside \mathcal{D} , we conclude that it is M alone who can make $\text{enc}(I)$ compatible with \mathcal{D} .

Drawbacks and improvements

In most cases, it would be useful if it was possible to display some unencrypted items with \mathcal{D} as well. However, we want to be able to distinguish between a document, which was originally delivered in plaintext form, and another document which was purchased in encrypted form and later decrypted. In particular, the decryption $D(\text{enc}(I), K)$ of $\text{enc}(I)$ should be distinguishable from any originally unprotected piece of data. Then, even if a hacker was able to decrypt $\text{enc}(I)$ (i.e. the key K was somehow revealed), this decrypted version would not be too useful because it would be rejected by the device \mathcal{D} .

One solution is to embed an 'information bit' b into I before encryption, thus labeling I as a protected document. For instance, if $b = 1$, then any \mathcal{D} would refuse to process the data when input in the plaintext form. The bit b can be embedded in I using some robust watermarking scheme, so that b cannot be removed or its value changed without also destroying the document (see f.ex. [2], [3], [6]). The device \mathcal{D} then checks the value of the watermark in I , and decides whether to reproduce I or not on the grounds of the value of b .

We must require that any key K is authorized by some trusted third party T . Otherwise, if a hacker H can access I which is decrypted but unreproducible with any legal device because of the watermark, he can easily sidestep the hindrance caused by b . Namely, re-encrypting I will do the trick: H can choose the encryption key, take the position of M and thus compute any device-dependent key β .

To prevent this possibility, M includes T 's signature for the encryption key K to the device-dependent key β in Step 2 of the protocol. In other words, β becomes a triple (x, y, z) , where x and y are as before, and $z = \text{sig}_T(K)$. When \mathcal{D} gets β

as input, it checks the validity of z before using K . Naturally, it must be assumed that hackers cannot obtain T 's signatures for their own keys.

There is still another major weakness in the system. Suppose that hacker H has got hold of a key K used for some I , and that some user B has acquired the same I . Thus, B has received the respective decryption key $\beta = (x, y, z)$. Now H and B can collude: any time the hacker is able to hack some item J , then he can re-encrypt it with K . Consequently, B can use the same β to decrypt J , without having to purchase a legal copy.

The obvious solution is to bind each key K to a specific item I . We propose a few methods. The simplest one is to set $z = \text{sig}_T(K \parallel I)$. The drawback is that then \mathcal{D} has to read all of I before it can determine whether z is valid. A more practical solution would be to divide I into blocks I_0, I_1, \dots, I_n , and encrypt each with a different key K_0, K_1, \dots, K_n . K_0 is the original key K , and each K_i is a function of K_{i-1} and a hash of I_{i-1} . In this case we have $z = \text{sig}_T(K_0 \parallel I_0)$, and \mathcal{D} can decide z 's validity right after reading I_0 .

A variant would be to set $z = \text{sig}_T(K \parallel w)$, where w is a watermark embedded in I prior to encryption. w could contain any kind of information on I , the purchase etc. The information bit b could be included in w as well. Depending on the placement of w in I , again \mathcal{D} must have read at least some of I before it can verify z .

The primary aim of the proposed content protection system is to prevent hackers from getting hold of unencrypted data items, and - if failing in this - secondary to minimize the usability of illegally decrypted data. The scheme does not have traitor tracing feature. This means that should we come short of both the goals, and illegal copies of I were made and distributed, there would be no way to find who is to blame.

A way to add traceability would be to make all the legal copies look different, i.e. uniquely fingerprint them. The devices \mathcal{D} could be equipped with an additional watermarking module and each I would be labeled before putting it out. Each device would have an unique watermarking pattern, and hence each copy of I would be different and distinguishable. The obvious weakness of this solution is that it relies quite heavily on the tamper-resistance of the device. One could argue, that if somebody can hack \mathcal{D} to obtain K and I , he would probably be able to pass by the watermarking module as well.

3 Encryption with Watermarking

In this chapter, we will describe a purchase protocol combining encryption with watermarking. Data encryption adds to the privacy of the parties, watermarking enables copyright protection and traitor tracing. We choose to use a watermarking scheme which is compatible with the compression procedure, since it is customary to compress any data prior to sending it over a transmission channel. Next we will discuss watermarking and combining it with compression, after which the scheme

with encryption and watermarking is presented. We assume that the subject of the purchase I is an image. A watermark, a bit sequence of length N , is denoted by w .

Combining watermarking with compression

In general terms, watermarking an image I means encoding the bits of a watermark w into I in some imperceptible way. A usual practice is to divide I in blocks of 8×8 pixels, and (pseudorandomly) choose the blocks in which the watermark will be embedded. Values of certain coefficients of I will be manipulated, and their absolute or relative values will then indicate the values of encoded bits of w .

Images are usually expressed by giving the gray-scale value(s) of each pixel. However, to achieve greater robustness and minimization of the computation time, watermarking is often performed in some transform domain instead of the spatial one. Namely, it is easier to predict the effects of compression (or some other manipulation) on the watermark if we work in the same domain as the manipulating algorithm.

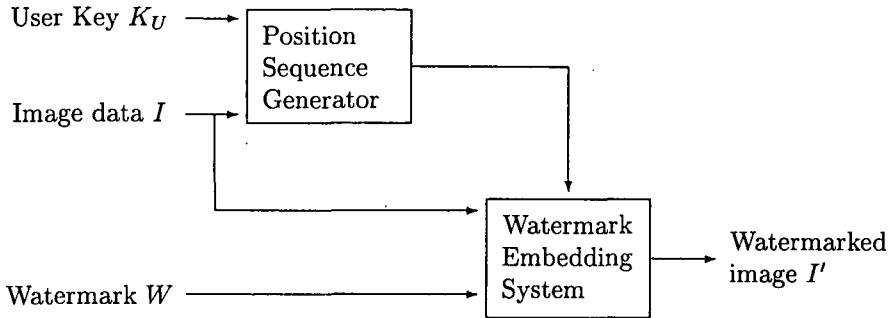
Compression algorithms make use of different transformations to separate the data into parts of different importance with respect to visual quality. *Discrete Cosine Transformation* (DCT) is an orthogonal transformation exploited by e.g. JPEG and MPEG algorithms. Its different basis vectors capture different features present in the input data. The effect is that the role of low frequency coefficients is emphasized, whereas most of the high frequency coefficients are small, and will be rounded off to zero during the compression. Therefore, placing the watermark in the low frequency DCT coefficients greatly adds to its robustness against compression. (For a throughout introduction on current watermarking schemes, see Chapter 6 in [3], or Chapter 8 in [2] on robust watermarking in general. A detailed description of the baseline JPEG can be found in [5].)

As changes in low frequency components easily become perceptual, various perceptual models, i.e. models imitating human visual system, are exploited in watermarking. One can use these models to compute so-called masking constraints, upper boundaries for the amounts a certain coefficient can be changed without causing visual effects. For more on the subject, see Chapter 7 in [2].

Many of the masking functions give the boundaries in the spatial domain instead of the frequency one. Often the problem has been solved by first embedding the watermark in the frequency domain and then cutting the visible changes in the spatial domain. The watermark remains imperceptible, but suffers in robustness. The framework by Pereira & al. in [4], however, enables strong watermarking in the frequency domain without violating the constraints in the spatial domain.

3.1 Combining watermarking with encryption

Whichever watermark embedding system or masking function is used, we can assume the outline of the procedure to be as follows:



The Position Sequence Generator is used to pseudorandomly select the pixel blocks of the image in which the watermark is placed. Hereafter, we use the word "image" as referring to a single 8×8 block, and "embedding a watermark" means encoding a single bit of the watermark in the block. This is natural, since embedding a longer watermark in a bigger image means just repeating this procedure for many enough blocks. The following notation is adopted:

- I = the DCT-coefs of an image block
- I' = the DCT-coefs of the image after watermarking
- w = the watermark
- E = encryption coefficients
- D = decryption coefficients.

For simplicity, we assume that all the variants above are real valued vectors of length 64, although most of the entries are zero for w , E and D . By $+$ we denote component-wise addition of two vectors.

Whichever the actual watermarking embedding system, *watermarking* means making imperceptible changes in some low frequency DCT coefficients of the image: $I' = I + w$. We note that encryption is realizable in the fashion of watermarking, by making *perceptible* changes in the coefficients: $enc(I) = I + E$. Here E 's non-zero entries are placed in the low frequency DCT coefficients, and they are large in magnitude compared with w .

Naturally, decryption reverses the effects of encryption in a straightforward way: $I = enc(I) + D$ where $D = -E$. However, the idea of the following protocol is to combine watermarking and encryption through 'imperfect decryption', that is by setting $D = w - E$. Then

$$\begin{aligned}
 enc(I) + D &= enc(I) + (w - E) \\
 &= I + E - E + w \\
 &= I + w \\
 &= I'.
 \end{aligned}$$

Clearly, decryption strips off most but not all of encryption, leaving I watermarked. Further, if w is unique, then so is I' .

Let again Merchant M and Buyer B be the parties of a purchase protocol. M has image I for sale, which he encrypts prior to setting it for distribution. B has a device \mathcal{D} to display the data. During the protocol, M delivers B a unique decryption vector D . As comparison between two decryption vectors D and D' gives away a lot of information on the respective watermarks w and w' , any two buyers of the same item could collude and easily destroy the watermarks, unless the decryption vectors were somehow protected. We solve the problem by adding a *device mask* as follows.

Connected to every device, there is a unique device key K_{dev} which determines a mask Dev . Dev is an integer vector, which the device will automatically subtract from the DCT-coefficients of any data prior to reproducing it. Therefore M adds the respective Dev to each decryption vector D :

$$enc(I) + D = I + E + (w + Dev - E) = I' + Dev.$$

It is important that the explicit value of K_{dev} remains unknown to everybody except for M , because the presence of a secret Dev in D makes comparisons between different decryption keys useless. However we assume that B has an index number k , with which B can enable M to find out the actual K_{dev} .

The main features of the protocol are as follows:

- *Purchase*: B sends the index k to M for computing K_{dev} . M returns B a unique decryption key $K_{D,B}$. Applying $K_{D,B}$ to $enc(I)$ using the device \mathcal{D} , B receives a copy of I with a unique watermark w_B .
- *Tracing*: Suppose B illegally redistributes his copy of I . He can be traced on the basis of the watermark w_B , which can be extracted only from the copies originating from his version of I .
- One encryption of I can be distributed to all buyers, but each decryption key is bound to a certain buyer with a certain device. The unique watermarking is forced to be done along the decryption.

The Protocol

We will adopt the following notation:

- | | | |
|-----------|---|---|
| Dev | = | the mask removed from any input data by \mathcal{D} on the basis of the key K_{dev} . |
| w_B | = | a unique watermark embedded in B 's copy of I . |
| K_E | = | the encryption key, on basis of which the encryption vector E is computed. |
| $K_{D,B}$ | = | the decryption key, from which the decryption vector D_B for buyer B is achieved. |

The protocol consists of a preliminary step (step 0), the purchase phase (steps 1 to 3), and step 4 where B 's device decrypts and reproduces the data.

0. M encrypts image I with a secret, symmetric key K_E . Encrypted image $enc(I)$ is set for distribution.
1. B gives M the index k for computing the key K_{dev} .
2. M computes K_{dev} on the basis of given k , chooses a watermark w_B for B , and computes a unique decryption key $K_{D,B}$ s.t. $D = Dev + w - E$.
3. M returns $K_{D,B}$ to B .
4. B applies $K_{D,B}$ together with the device key K_{dev} to $enc(K)$.

In the last step,

$$\begin{aligned} enc(I) \rightarrow enc(I) + D &= I + E - E + Dev + W \\ &= I' + Dev, \end{aligned}$$

and further

$$I' + Dev \rightarrow I' + Dev - Dev = I'.$$

Hence the result of \mathcal{D} 's computations is the watermarked image I' .

We have not specified the correspondences $K_E \sim E$, $K_D \sim D$, or $K_{dev} \sim Dev$. Use of the keys is necessary, since the actual vectors D and Dev are too long and too many to be transmitted as such (even though they mostly consist of zeros). A mapping to pack the information is needed. Possible solutions are many, as an example we give one.

We have thought of D and Dev as 64-dimensional integer vectors. Let us present a vector as a concatenation of the binary presentations of its entries, and let N be an integer such that for every entry i in D or Dev , $|i| \leq |N|$. The length of the binary presentation is then $64 \cdot \log_2(2N)$. For $N = 2^{15}$ this equals 1024. We can establish a one-to-one correspondence between 1024-long binary vectors and the elements of the group \mathbb{Z}_p^* , where $|p| = 1024$. Therefore, each vector D or Dev can be presented as an element of \mathbb{Z}_p^* .

Note that most of the entries of the vectors are zeros, as it is enough to mask/encrypt about five to twenty most significant of them. Thus, we can cut the extra zeros by setting for example $D, Dev \in [-N, N]^{20}$. Then, for $N = 2^{15}$, $20 \cdot \log_2(2N) = 320$ and thus the vectors can be expressed as elements of \mathbb{Z}_p^* , where $p = |320|$ only.

In the previous scheme we assumed there is a mapping $k \rightarrow K_{dev}$, which remained unknown to B but could be found out by M . In this case, it could be f.ex. a permutation on \mathbb{Z}_p^* . In the following chapter, we will re-examine the concepts of and relations between k , K_{dev} and Dev .

3.2 On the device key K_{dev}

Consider the following scheme: buyers B and B' with devices \mathcal{D} and \mathcal{D}' resp., both purchase an encrypted item $enc(I) = I + E$ from the merchant M . The following communication takes place:

$$\begin{aligned} M \rightarrow B : \quad K_{D,B} \sim D_B &= -E + Dev_{\mathcal{D}} + w_B \\ M \rightarrow B' : \quad K_{D,B'} \sim D_{B'} &= -E + Dev_{\mathcal{D}'} + w_{B'} \end{aligned}$$

Here is a chance for B and B' to collude. Subtracting one decryption vector from the other, they learn $S = Dev_{\mathcal{D}} - Dev_{\mathcal{D}'} + w_B - w_{B'}$. Here $w_B - w_{B'}$ is small, thus $S \approx Dev_{\mathcal{D}} - Dev_{\mathcal{D}'}$. Now, if B buys another image $enc(J) = J + E_J$, then B' can use the corresponding decryption vector $D_B^J = -E_J + Dev_{\mathcal{D}} + w_B$ by computing

$$\tilde{D}_{B'}^J = D_B^J - S = -E_J + Dev_{\mathcal{D}} + \epsilon,$$

where $\epsilon = w_B' + w_{B'} - w_B$ is a small error. Decryption of $enc(J)$ with the new vector $\tilde{D}_{B'}^J$, using the device \mathcal{D}' yields $J' \rightarrow J + Dev_{\mathcal{D}} + \epsilon \rightarrow J + \epsilon$. Thus, B' can use his own device to reproduce B 's copy of J with only small distraction.

The above scheme suggests that the device mask Dev should not be fixed, but different for each I . In our model, Dev is deterministically computed from a device key K_{dev} (see the end of the previous chapter). Thus, what we need is a method to generate keys K_{dev} . As the computation $K_{dev} \rightarrow Dev$ is reversible, K_{dev} must remain unknown to B .

Relying on the tamper-resistance of \mathcal{D} , the keys could be generated within the device by some function $f^n(k) = K_{dev}^n$, for $n = 1, 2, \dots$ etc. The merchant would be able to compute Dev if he was given the pair (k, n) instead of the index k only. However, the system with indices and secret generating functions seems somewhat impractical, because duplex communication between B and \mathcal{D} is needed, as well as an active third party with the knowledge of $k \sim K_{dev}$ correspondences and the functions f .

To avoid these difficulties we take a new starting point: allowing M to take part in the generation of K_{dev} . If M is able to compute K_{dev} on his own, then the role of index k shrinks into tying K_{dev} to \mathcal{D} . If M can actually *decide* the value of the key (and thereby of the mask) used in decryption, then M can as well give the value of the decryption key and the mask together. In other words, M can provide B with a key K , which corresponds with the vector $D_B - Dev$, instead of giving $K_{D,B}$ and K_{dev} separately.

Let us discuss options of carrying out the above scenario. Let M provide \mathcal{D} with a seed d to generate K_{dev} , as a function of both I and k , for example. M generates d from I , and computes $f(k, d) = K_{dev} \sim Dev$. If M sends d to B together with the decryption key (step 3), then \mathcal{D} can compute K_{dev} too. The problem is that so can B , unless the function f is kept secret from B (but it has to be available to \mathcal{D} , which in turn again would complicate the system).

Function f is not needed, if M decides the value of K_{dev} and sends it to B as such. However, if K_{dev} has the value of an element in \mathbb{Z}_p^* , then the correspondence

$\mathbb{Z}_p^* \sim \{0, 1\}^{lp}$ between the key and the mask must remain unknown to B (but accessible to D). If D possessed a public key pair (s_D, t_D) , then M could protect K_{dev} from B by encrypting it with t_D before handing it out. D would still be able to find out K_{dev} , since it has access to s_D .

The scheme with D possessing a public key pair (s_D, t_D) where s_D is accessible to D only (c.f. the scheme in Chapter 2) seems useful. We can use the Diffie-Hellman protocol to generate the key K_{dev} as follows. M creates an ephemeral key pair (s_M, t_M) , and performs the D-H protocol to obtain $K_{dev} = t_D^{s_M}$. Then he can further compute Dev , and compute the decryption vector $D_B = -E + Dev + w_B$. Now M sends B both $K_{D,B}$ and t_M . Given these, D can decrypt $enc(I)$, since $enc(I) + D_B = (E + I) + (-E + Dev + w_B) = I + Dev + w_B = I' + Dev$, where $Dev = t_M^{s_D}$. On the other hand, on basis of the given information, B cannot learn and remove Dev , as he does not know s_D which is needed to compute K_{dev} .

Assume K_{dev} is generated as above. M wants to give B a key K such that $K \sim D_B - Dev = -E + w_B$ (we assume that he can easily compute the target value K once he knows $D_B - Dev$). To give $D_B - Dev$ with a single key, we can proceed as follows.

1. M computes the value of $K \sim Dev + D_B$.
2. M generates a random public-key pair (s_M, t_M) , and computes $K_{dev} = t_D^{s_M}$.
3. M computes the difference $\Delta = K - K_{dev}$ and sends t_M and Δ to B .
4. D computes $K_{dev} = t_M^{s_D}$, and further $K = \Delta - K_{dev}$.

As B knows only that Δ is the difference between K and K_{dev} , he cannot find out either of these values because he does not know s_D . The key K is computed in each end of the transmission channel, but not transmitted at all.

The above method can be applied even if K_{dev} was generated in some other manner, as long as M can find out the target value $K \sim D_B - Dev$ on his own. Then the difference between a random key $t_D^{s_M}$ and the target is computed, and t_D and the corrective key (Δ above) are given to B .

References

- [1] B. den Boer, Diffie-Hellman is as Strong as Discrete Log for Certain Primes, *Proceedings of CRYPTO'88*, LNCS 403, Springer-Verlag, 1988, pp. 530-539.
- [2] I. Cox, M. Miller and J. Bloom, *Digital Watermarking*, Academic Press, San Francisco 2002.
- [3] S. Katzenbeisser and F. Petitcolas (ed.), *Information Hiding Techniques for Steganography and Digital Watermarking*, Artech House, London 2000.
- [4] S. Pereira and T. Pun, Optimal Transform Domain Watermark Embedding Via Linear Programming. *Signal Processing* 81, No. 6 2001, pp. 1251-1260.

- [5] K. Wallace, The JPEG still picture compression standard. *Communications of the ACM* 34, No. 4 1991, pp. 30-40.
- [6] J. Zhao and E. Koch, Embedding Robust Labels Into Images For Copyright Protection. *Proceedings of the International Congress on Intellectual Property Rights for Specialized Information, Knowledge and New Technologies*, Vienna 1995.

Evaluation of a Fully Automatic Medical Image Registration Algorithm Based on Mutual Information*

Attila Tanács[†] and Attila Kuba[‡]

Abstract

Registration is a fundamental task in image processing. Its purpose is to find a geometrical transformation that relates the points of an image to their corresponding points of another image. Many registration algorithms have been proposed in the past decade. We present a fast, fully automatic algorithm that is capable of solving rigid-body registration of 3D images of the human brain where the images are taken by different imaging devices. We joined the Retrospective Registration Evaluation Project conducted by Vanderbilt University, USA. The evaluations of our results show that our method has the potential to produce satisfactory results, but visual inspection is necessary to guard against large errors.

Keywords: registration problem; automatic multimodal registration; registration accuracy;

1 Introduction

There is an increasing number of applications that require accurate aligning of one image with another taken from different viewpoints, by different imaging devices, or at different times. The geometrical transformation is to be found that maps a *floating image data set* in precise spatial correspondence with a *reference image data set*. This process of alignment is known as *registration*, although other words, such as *co-registration*, *matching*, and *fusion*, are also used. Examples of systems where image registration is a significant component include aligning medical images from different medical modalities for diagnosis, matching a target with a real-time image of a scene for target recognition, monitoring global land usage using satellite images, and matching stereo images to recover shape for autonomous navigation [1, 8].

*This work was supported by OTKA T023804 and FKFP 0908/1999 Grants.

[†]Department of Foundations of Computer Science, University of Szeged, 6720 Szeged, Árpád tér 2, Hungary, e-mail: tanacs@inf.u-szeged.hu

[‡]Department of Applied Informatics, University of Szeged, 6720 Szeged, Árpád tér 2, Hungary, e-mail: kuba@inf.u-szeged.hu

In this paper we focus on medical image registration which has a wide range of applications including combining information from multiple imaging modalities e.g., when relating functional information from nuclear medicine images to anatomy delineated in high-resolution MR or CT images, monitoring changes in size, shape, or image intensity over time intervals ranging from few seconds to even months or years, relating preoperative images and surgical plans to the physical reality of the patient in the operating room during image-guided surgery or during radiotherapy, and relating an individual's anatomy to a standardized atlas.

The registration technique for a given task depends on the knowledge about the characteristics of the type of variations. Registration methods can be viewed as different combinations of choices for the following four components [1]:

- *Search space* is determined by the type of transformation we have to consider, i.e., what is the class of transformations that is capable of aligning the images. Some widely used types are *rigid-body*, when only translations and rotations are allowed, *affine*, which maps parallel lines to parallel lines, and *nonlinear*, which can transform straight lines to curves.
- *Feature data set* describes what kind of image properties are used in matching. Features can be geometrical, e.g., automatically or manually selected landmark points, lines, and/or surfaces or the image intensity values can be used directly.
- *Similarity measure* is a function of the transformation parameters which shows how well the floating and the reference image fit. The task of registration is to optimize this function.

In case of geometrical features this is usually a distance measure. When image intensity values are used, correlation, functions based on image intensity differences, or intensity similarity measures can be applied.

- *Search strategy* determines what kind of optimization method to use. Except for geometric features, where a direct solution of the problem might exist, an iterative approach is necessary.

In this paper we propose a fully automatic, iterative registration method that is capable of finding rigid-body transformations to align images from the same or different modalities (i.e., taken by the same or different imaging devices). Intensity similarity measures based on mutual information are used.

2 Methods

We follow the notations of [7]. Let X denote the object to be imaged, and let A and B be 3D images of X taken by the same or different imaging devices. The images usually have different fields of view, thus the domains Ω_A and Ω_B will be different:

$$\begin{aligned} A &: x_A \in \Omega_A \mapsto A(x_A), \\ B &: x_B \in \Omega_B \mapsto B(x_B). \end{aligned}$$

$A(x_A)$ and $B(x_B)$ are referred to as the intensity values at spatial positions x_A and x_B , respectively. Intensity values represent some kind of measurement of the material in spatial positions of X , such as attenuation of X-ray beams in case of Computed Tomography (CT), changes in states of protons under changing the magnetic field properties in Magnetic Resonance Imaging (MRI), or distribution of nuclear tracers in case of Positron Emission Tomography (PET) and Single Photon Emission Computed Tomography (SPECT).

As the images A and B represent the same object X , there is a relation between the spatial locations in A and B . Position $x \in X$ is mapped to x_A in image A , and to x_B in image B . The registration process involves recovering the spatial transformation T which maps x_B to x_A over the entire domain of interest, which is the overlapping portion of the domains. This overlapping portion depends on the images A and B and on the spatial transformation T :

$$\Omega_{A,B}^T = \{x_A \in \Omega_A | T^{-1}(x_A) \in \Omega_B\}.$$

The medical images are discrete, they sample the object at a finite number of points. Taking this into account, we can define the domain Ω in the following way:

$$\Omega := \tilde{\Omega} \cap \Gamma_\zeta$$

where $\tilde{\Omega}$ is a bounded continuous set defining the volume of the patient imaged, and Γ is an infinite discrete sampling grid, which is characterized by the anisotropic sample spacing $\zeta = (\zeta^x, \zeta^y, \zeta^z)$. Sample spacing can be different for different images. These grid positions and the corresponding sample values together are referred to as voxels. For any given T , the intersection of discrete domains Ω_A and Ω_B might be the empty set, when no sample points will exactly overlap. To overcome this, we have to resample image intensities of image B in Ω_A . The simplest resampling method is to select the intensity value of the closest grid position of Ω_B . Linear or more complex interpolation methods can also be used. Let \mathcal{T} denote the transformation that maps both the position and the associated intensity value at that position, and B^T the resampled image B .

The selection of the similarity measure is probably the most crucial part of a registration algorithm. We need a function which optimally has one global optimum at perfect alignment, has no local optimums, and is "smooth enough" to find this optimum fast. Practically it is very hard, or even impossible to find such a similarity measure, especially when the images are taken by different imaging devices. Many similarity measures were proposed in the past decade. We chose the measures based on the mutual information of the images proposed by Collignon et al. [4] and Wells et al. [12], and on the normalized mutual information of the images proposed by Studholme et al [11].

Both measures utilize the entropy of image A ,

$$H(A) = - \sum_a p_A^T(a) \cdot \log p_A^T(a),$$

the entropy of image B ,

$$H(B) = - \sum_b p_B^T(b) \cdot \log p_B^T(b),$$

and the joint entropy of images A and B ,

$$H(A, B) = - \sum_a \sum_b p_{AB}^T(a, b) \cdot \log p_{AB}^T(a, b),$$

where p_A and p_B are the histograms, and p_{AB} is the co-occurrence matrix of the intensity values of images A and B . Mutual information is computed as

$$MI(A, B) = H(A) + H(B) - H(A, B),$$

and the normalized mutual information as

$$NMI(A, B) = \frac{H(A) + H(B)}{H(A, B)}.$$

We found that when mutual information is calculated over the overlapping domain $\Omega_{A,B}^T$, the failure rate is high [11]. We decided to use the whole Ω_A instead, in case of this measure, which solved the problem.

To speed up the registration process and to avoid falling into a local optimum, we use the Laplacian multiresolution pyramid representation of the images [2]. The search starts at the coarsest level. When an optimum is found, the result is propagated to the next, finer level. For the registration task of this project, we generate two new coarser pyramid levels.

We use Powell's direction set, iterative, nonlinear optimization algorithm to find the optimum of the similarity measure [10]. This method requires evaluating the similarity measure value for given transformation parameters only, no gradient or other information is necessary. The most time consuming part of the method is the evaluation itself, so it is crucial to avoid any unnecessary computations.

When resampling, we can take advantage of the fact that the transformation we are looking for is a linear one, which means that parallel lines, e.g., rows and columns remain parallel lines after applying the transformation. Using a general 3D line drawing algorithm [6], the resampling can be done using additions only, no multiplications are necessary. We use no interpolation of intensity values, we select the value of the nearest neighbor.

When the image sizes are no larger than 256 voxels, we can represent floating point numbers as 32-bit integers. Thus we have 1 sign bit, 9 bits for the integer part, and 22 bits for the fraction part. The precision of this representation is worse than that of the built-in floating point types, but is still good enough. We performed

numerical simulations to check the inaccuracy. Rigid-body transformations were generated randomly and applied to the points of a grid of size $256 \times 256 \times 25$, with grid spacing of 1.25, 1.25, 4.00, respectively. Both real floating point and integer representations of reslicing methods were used and the maximum distance of the transformed points was calculated. The comparison showed that the maximum difference between spatial locations was about 0.02 voxels. For this price we get dramatic speed boost.

During resampling, we calculate probabilities p_A^T , p_B^T , and p_{AB}^T for each intensity value. The calculation of $MI(A, B)$ can be made faster as follows. By definition,

$$\begin{aligned} MI(A, B) &= - \sum_a p_A^T(a) \cdot \log p_A^T(a) - \sum_b p_B^T \cdot \log p_B^T(b) + \\ &\quad \sum_a \sum_b p_{AB}^T(a, b) \cdot \log p_{AB}^T(a, b) \\ &= \sum_a \sum_b (p_{AB}^T(a, b) \cdot \log p_{AB}^T(a, b) - p_A^T(a) \cdot \log p_A^T(a) - \\ &\quad p_B^T \cdot \log p_B^T(b)). \end{aligned}$$

Since the marginal probability distributions can be calculated from the joint probability distribution,

$$\begin{aligned} p_A^T(a) &= \sum_b p_{AB}^T(a, b), \\ p_B^T(b) &= \sum_a p_{AB}^T(a, b), \end{aligned}$$

mutual information can be calculated as

$$MI(A, B) = \sum_a \sum_b p_{AB}^T(a, b) \cdot (\log p_{AB}^T(a, b) - \log p_A^T(a) - \log p_B^T(b)).$$

The probabilities can have a value between 0 and 1, thus instead of calculating logarithmic values, we can use a precalculated lookup table, say the size of 10000 elements.

Real medical images can usually have intensity values ranging from -1000 to 4000. It means that the joint probability distribution table should have $5000 \cdot 5000 = 25000000$ elements, which is not feasible. That is why we scale intensity values so as to be in the $[0, 63]$, $[0, 127]$, or $[0, 255]$ ranges before registration.

Algorithm 1 summarizes the main steps of the method we applied.

3 Evaluation of the registration method

It is necessary to measure the degree of alignment in order to determine whether a given registration technique is adequate for a given problem. The alignment need not be perfect, but the error must be below a certain threshold. The similarity

Algorithm 1: Registration algorithm

Input: Two 3D images A and B with known dimensions and sample spacing
Output: Rigid-body transformation $optT$ that maximizes the mutual information or the normalized mutual information of images A and $optT(B)$

```

begin
1  scale intensity values of both images to be in  $[0, 127]$ ;
2  generate  $A_l$  and  $B_l$ , the multiresolution Laplacian pyramid representation of the images ( $l = 0, \dots, L$ );
3  let  $T$  be the identity transformation;
4   $optT = T$ ;
5  for each pyramid level  $l$  from coarsest to finest do
6     $optI = MI(A_l, optT(B_l))$ ;
7    repeat
8       $T = optT$ ;
9      make a change to  $T$  (Powell's method);
10      $m = MI(A_l, T(B_l))$ ;
11     if  $m > optI$  then
12        $optI = m$ ;
13        $optT = T$ ;
14     endif
15   until  $optT$  was not changed;
16   endfor
end

```

measure cannot be used to judge this, since it is not guaranteed that it reaches its global optimum at perfect alignment. An other method, visual inspection plays an important role. When a suitable interactive image viewing software is available, the human visual system can detect errors greater than 2 mm for CT to MR, and 4 mm for PET to MR registration [5, 15]. Although visual inspection is always necessary, since the automatic methods occasionally might fall into a nonglobal optimum producing a bad result without any warnings, a more accurate evaluation procedure is necessary. An overview of such procedures can be found in [7].

To evaluate our registration method, we joined the Retrospective Registration Evaluation Project of Vanderbilt University, USA in 1999 [13]. The objective of that project was to perform blinded evaluation of retrospective image registration techniques using a prospective, marker-based registration method as a gold standard. A gold standard is a system whose accuracy is known to be high. A fiducial marker system can serve as an excellent gold standard for rigid registration, since some of these systems can provide submillimetric accuracy. The primary disadvantage is the high invasiveness i.e., bone-implanted markers [9]. In order to ensure blindness, all retrospective registrations were performed by participants who had no knowledge of the gold-standard until after their results had been submitted.

Image volumes of three modalities: X-ray computed tomography (CT), magnetic resonance (MR), and positron emission tomography (PET) were obtained from patients undergoing neurosurgery at Vanderbilt University Medical Center, on whom bone-implanted markers were mounted. These volumes had all traces of the markers removed and were provided to project collaborators outside Vanderbilt, who then performed registration on the volumes. The investigators communicated their results to Vanderbilt, where the accuracy of each registration was evaluated.

Two registration tasks were evaluated: CT to MR and PET to MR, and these tasks were broken into subtasks according to the type of MR and to whether or not the MR image was corrected (rectified) for geometrical distortion [3]. The image data set of nine patients were used, seven of which contained both CT and MR, and seven with both PET and MR.

The CT volumes have a resolution of 512 pixels in the x and y directions, and have between 28 and 34 slices in the z direction. The voxel size is 0.65 mm in x and y , and 4.0 mm in z . The MR volumes have a resolution of 256 pixels in the x and y directions, and have 20 to 26 slices. The voxel size is between 1.25 and 1.28 mm in the x and y directions, and 4.0 mm in z . The PET volumes have 15 slices with a resolution of 128 pixels in the x and y directions. The voxel size is 2.59 mm in x and y , and 8.0 mm in z .

At Vanderbilt, in collaboration with a neurological and a neurosurgical expert, a set of VOIs (Volume of Interest) representing areas of neurological and/or surgical interest was manually segmented within one of the MR image volumes for each patient. An estimate of the accuracy of the retrospective registration at the position of each VOI is computed as follows. The centroid pixel of the VOI is found, and its position is converted from voxel index to a millimetric position c in the reference volume using the known size for the image volume. Let T_G denote the gold-standard rigid-body transformation, and T_R the result of the retrospective registration algorithm. The point c' in the floating image is defined so that c is the mapping of c' under the gold-standard transformation,

$$c = T_G(c').$$

Thus,

$$c' = T_G^{-1}(c).$$

The point c'' in the reference image is defined as the mapping of c' under the retrospective transformation,

$$c'' = T_R(c').$$

The error of the retrospective registration at the anatomical position of the VOI is defined as the Euclidean distance between the registered target position of the retrospective method and that of the gold standard, $\|c'' - c\|$.

4 Results

The results of the project were published in [13] and [14]. Since we joined the project later, our results were not included in those papers. Here we compare our

results against those evaluated earlier.

Ten groups of investigators applied 14 techniques to solve the registration tasks. The techniques were divided into two groups. Any technique which performs registration by making use of a relationship between voxel intensities within the images is referred to as volume based, and any technique which works by minimizing a distance measure between two corresponding surfaces in the images to be matched is referred to as surface based. Six of the 14 techniques were volume based and eight were surface based. Our methods can be classified as volume based ones.

Before the evaluation of our results, we visually inspected the quality of registration. When the normalized version of the mutual information was used, all registration results were visually acceptable. In case of mutual information, for the CT to MR task, all 41 results were visually acceptable. In case of PET to MR, for five image pairs the results of registration was visually misregistered. These pairs were PET to MR PD, MR T1, MR PD rectified of Patient 6 and PET to MR T1, and MR T2 rectified of Patient 8. The other 30 results were visually acceptable. In spite of these clear misregistrations, all results were submitted for evaluation to Vanderbilt University.

Table 1 shows the statistics of registration errors for the groups of algorithms and the rankings of our methods out of the 16 competing methods.

Modality	Surface based mean error (std.dev.)	Volume based mean error (std.dev.)	Our MI mean error (ranking)	Our NMI mean error (ranking)
CT-T1	5.7 (7.8)	2.9 (2.4)	1.6 (#2)	2.3 (#7)
CT-PD	5.8 (8.0)	2.9 (2.5)	2.2 (#2)	1.8 (#1)
CT-T2	6.3 (7.9)	2.4 (1.4)	2.0 (#5)	2.0 (#3)
CT-T1 rect.	6.1 (8.3)	2.0 (2.5)	1.7 (#5)	2.2 (#7)
CT-T2 rect.	5.7 (7.8)	1.8 (2.0)	1.4 (#3)	2.3 (#7)
CT-PD rect.	6.1 (7.6)	2.1 (1.6)	1.7 (#4)	2.4 (#7)
PET-T1	3.9 (2.0)	3.5 (2.1)	5.3 (#9)	3.0 (#2)
PET-T2	4.4 (2.1)	3.6 (1.9)	3.8 (#7)	3.5 (#4)
PET-PD	4.3 (2.6)	4.0 (2.7)	4.4 (#7)	4.2 (#10)
PET-T1 rect.	3.9 (2.3)	2.7 (1.4)	3.8 (#12)	2.7 (#3)
PET-T2 rect.	3.9 (2.0)	3.5 (1.7)	3.9 (#10)	3.3 (#5)
PET-PD rect.	3.9 (2.3)	3.5 (2.4)	4.8 (#10)	3.0 (#2)

Table 1: Mean and standard deviation of registration errors. Note that the ranking of our methods is based on the median errors of the registration methods, as it is published in [13].

5 Discussion

The results show that in case of CT to MR registration task, both of our methods produce acceptable results. For PET to MR problems, the MI method tends to fail (five failures out of 35 cases), and produces average results. The NMI method gives stable results and ranks high among the competing algorithms.

The running time was about 30-120 seconds on a 800 Mhz Pentium-III PC. More detailed results of the evaluation of our methods can be found at <http://www.vuse.vanderbilt.edu/~images/registration>.

6 Conclusion

We presented a registration algorithm, which can be successfully used to align 3D medical images from different imaging modalities. The algorithm is fully automatic, needs no user interaction. However, before using the optimal transformation determined by the algorithm, it is necessary to visually inspect it to sort out possible misregistrations.

Acknowledgements

The images and the standard transformations were provided as part of the project, "Evaluation of Retrospective Image Registration", National Institutes of Health, Project Number 1 R01 NS33926-01, Principal Investigator, J. Michael Fitzpatrick, Vanderbilt University, Nashville, TN.

References

- [1] Brown, L.G.: A survey of image registration techniques. *ACM Computing Surveys* **24** (1992) 325-376
- [2] Burt, P.J., Adelson, E.H.: The Laplacian Pyramid as a Compact Code. *IEEE Trans. on Communications* **31** (1983) 532-540
- [3] Chang, H., Fitzpatrick, J.M.: A technique for accurate magnetic resonance imaging in the presence of field inhomogeneities. *IEEE Transactions on Medical Imaging* **11** (1992) 319-329
- [4] Collignon, A., Maes, F., Delaere, D., Vandermeulen, D., Suetens, P., Marchal, G.: Automated multi-modality image registration based on information theory. In *Proceedings of Information Processing in Medical Imaging* (1995) 263-274
- [5] Fitzpatrick, J.M., Hill, D.L.G., Shyr, Y., West, J.B., Studholme, C., Maurer, C.R.Jr.: Visual assesment of the accuracy of retrospective registration of MR and CT images of the brain. *IEEE Transactions on Medical Imaging* **17** (1998) 571-585

- [6] Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F.: *Computer Graphics — Principles and Practice*. Addison-Wesley Publishing Company, Reading, Massachusetts (1991)
- [7] Hajnal, J.V., Hill, D.L.G., Hawkes, D.J. (eds.): *Medical Image Registration*. CRC Press (2001)
- [8] Maintz, J.B.A., Viergever, M.A.: A survey of medical image registration. *Medical Image Analysis* **2** (1998) 1–36
- [9] Maurer, C.R., Fitzpatrick, J.M., Wang, M.Y., Galloway, R.L., Maciunas, R.J., Allen, G.S.: Registration of head volume images using implantable fiducial markers. *IEEE Trans. on Medical Imaging* **16** (1997) 447–462
- [10] Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.: *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition (1992)
- [11] Studholme, C., Hill, D.L.G., Hawkes, D.J.: An overlap invariant entropy measure of 3D medical image alignment. *Pattern Recognition*, **32** (1999) 71–86
- [12] Wells, W.M.III, Viola, P., Kikinis, R.: Multi-modal volume registration by maximization of mutual information. In *Medical Robotics and Computer assisted Surgery*, Wiley-Liss, New York (1995) 155–162
- [13] West, J.B., Fitzpatrick, J.M., et al.: Comparison and evaluation of retrospective intermodality brain image registration techniques. *Journal of Computer Assisted Tomography* **21** (1997) 554–566
- [14] West, J.B., Fitzpatrick, J.M., et al.: Retrospective Intermodality Registration Techniques for Images of the Head: Surface-Based Versus Volume-Based. *IEEE Trans. on Medical Imaging* **18** (1999) 144–150
- [15] Wong, J.C.H., Studholme, C., Hawkes, D.J., Maisey, M.N.: Evaluation of the limits of visual detection of image misregistration in a brain fluorine-18 fluorodeoxyglucose PET-MRI study. *European Journal of Nuclear Medicine* **24** (1997) 642–650

A Graphical User Interface for Evolutionary Algorithms*

Zoltán Tóth[†]

Abstract

The purpose of *Generic Evolutionary Algorithms Programming Library* (*GEA*¹) system is to provide researchers with an easy-to-use, widely applicable and extendable programming library which solves real-world optimization problems by means of evolutionary algorithms. It contains algorithms for various evolutionary methods, implemented genetic operators for the most common representation forms for individuals, various selection methods, and examples on how to use and expand the library. All these functions assure that *GEA* can be effectively applied on many problems. *GraphGEA* is a graphical user interface to *GEA* written with the GTK API. The numerous parameters of the evolutionary algorithm can be set in appropriate dialog boxes. The program also checks the correctness of the parameters and saving/restoring of parameter sets is also possible. The selected evolutionary algorithm can be executed interactively on the specified optimization problem through the graphical user interface of *GraphGEA*, and the results and behavior of the EA can be observed on several selected graphs and drawings. While the main purpose of *GEA* is solving optimization problems, that of *GraphGEA* is education and analysis. It can be of great help for students understanding the characteristics of evolutionary algorithms and researchers of the area can use it to analyze an EA's behavior on particular problems.

1 Introduction

Evolutionary algorithms (EAs for short) are general purpose function optimization methods that search for optima by making potential solutions (*individuals*) compete for survival in a *population*. The better a potential solution is, the better chance it has to survive. The individuals are represented by means of a predefined data structure (*genotype*), and the evaluation considers the performance of the individual in its current environment (*phenotype*). The search space is explored by modifying

*This work was supported by the grants of the German Academic Exchange Service (DAAD)

[†]Institute of Informatics, University of Szeged, Árpád tér 2, H-6720 Szeged, Hungary. Now visiting Department of Computer Science 2: Programming Systems, Friedrich-Alexander University of Erlangen-Nuremberg, Martensstr. 3, D-91058 Erlangen, Germany.
e-mail: zntoth@inf.u-szeged.hu

¹The project's home page can be found at <http://gea.ztoth.net>

the genotypes by *genetic operators* observed in nature: generally *mutation* and *recombination* [15, 22, 32].

Evolutionary algorithms have (among others) the following two advantages over other optimization methods: first, in most cases they converge to global optima, and second, the usage of the black-box principle (which only requires knowledge of a function's input and output to perform optimization on it) makes them easily applicable to functions whose behavior is too complex to handle with other methods. The huge amount of practical applications presented on numerous conferences show that EAs represent a relatively new and important group of function optimization methods. Nevertheless, being stochastic processes, it is hard to understand the functioning of a particular algorithm and build a suitable model of it. An even more difficult problem is to choose the optimal algorithm and determine the values of its parameters for a given problem or problem class.

Visualizing the interiors of an algorithm can be a great help in the *understanding* of its inner processes and behavior. For example, it is very easy to see the effects of a parameter or a selection method on the diversity of the population in the different phases of the evolution process.

The visualization of evolutionary algorithms is useful in *education*, too. Not just because it is much easier to fascinate students with a nice and handy graphical user interface, but also because they can become acquainted with the most important features of evolutionary computation. They can experience with different parameter settings and see that the changes in the behavior of the process are really those which they have heard of or read in the literature.

The purpose of this paper is to present a tool for visualizing evolutionary algorithms: the *GraphGEA* program. *GraphGEA* is a graphical user interface (GUI) to the *Generic Evolutionary Algorithms Programming Library (GEA)* [36]. *GEA* is an easily applicable and extendible evolutionary programming tool written in the C++ programming language. By interacting with the evolution process running in the background as its child process, the GUI shows the course and the status of the optimization in various configurable visualization windows. *GraphGEA* can be easily extended with new methods showing the interiors of the optimization, for these methods are realized as plug-ins of the system. The communication is implemented by means of the so-called pipe mechanism and UNIX IPC (inter-process communication).

Last but not least, an evolution process can have a great many parameters, the values of which are usually strongly interconnected or dependent. The *GraphGEA* program can just be used to manage optimization projects, for it assures that all necessary parameters of the selected algorithm and representation of individuals are correctly set.

In the following, Section 2 offers a short overview of evolutionary algorithms. The presented systems use a special data structure to hold the parameters of the evolution process, this data structure is presented briefly in Section 3. Section 4 deals with the *GEA* system: the class hierarchy, the functioning of the various evolutionary algorithms, the selection methods and the genetic operators are described. Section 5 presents the *GraphGEA* program with a detailed description of the user

interface and the visualization tools. In Section 6, some references to and comparisons with the related work can be found. Finally in Section 7 a summary of the work is given.

2 Evolutionary Algorithms

In this section an overview of evolutionary algorithms is given, focusing on details that are important for the *GEA* and *GraphGEA* systems.

Evolutionary algorithms (EAs) are general purpose function optimization methods which use the ‘survival-of-the-fittest’-model known from nature [8]. In this model, *individuals* compete for resources in an environment, and *selection* assures that individuals which are better suited for the given environment will produce more offspring. Thus the preservation of good attributes is guaranteed.

Unlike most optimization methods, EAs consider several potential solutions at a time. These potential solutions, called *individuals* from now on, form a *population*. The individuals interact with each other, thus they create new individuals to form a *new generation*.

An individual of the population is represented with a sort of data structure. The most common representation forms for individuals are *bit-string* and *real vector*. Each element of the vector is called a *gene*. The chain of genes is also called a *chromosome*. The values in it are the individual’s *genotype*. The appearance of an individual – which can be e.g. a permutation of certain numbers – is called *phenotype*. Evolutionary algorithms work on the level of the genotype, which means that they modify the encoded form of individuals. When *evaluating* an individual in its current environment, its phenotype is considered. The result of the *evaluation* is the *fitness value*, a specified extremum of which has to be found by the evolutionary algorithm. This fitness value is considered when performing *selection*.

The creation of new individuals is implemented by applying certain *genetic operators* on the selected parents. The most common genetic operators are *reproduction*, *mutation* and *recombination*. Reproduction and mutation are unary operators. Reproduction simply copies the individual into the new generation, while mutation modifies its argument by randomly changing each gene of it with a certain probability. Recombination takes two or more individuals and creates new ones by exchanging parts of their gene-chains. Each genetic operator is applied with a certain probability. However, sometimes one operator is more efficient than the others and it is not easy (or at least it requires experiment) to set the probabilities correctly at the start of an evolution process. Davis offers a solution to this problem: let’s change the probabilities dynamically during the evolution process by observing the effectiveness of the operators. He calls this method the *adaptation of operator probability* [9].

Generally, the *procedure of an evolutionary algorithm* is the following: the structures in the initial population can be generated randomly or, if an initiative solution is known, then that one can be used with random modifications. Then the individuals are evaluated and new generations are created until a termination condition

is satisfied, which, in the simplest cases, is reaching a certain generation number or the stagnation of the best individual's fitness value. The generation of the new population is absolutely algorithm-dependent, so these methods will be discussed at the specification of the algorithms.

Several kinds of evolutionary algorithms are known, the most important ones of which are *genetic algorithms* (GAs) [10, 15] and *evolution strategies* (ESs) [31]. They were developed independently in the 1970s: GAs were introduced by John Holland and analyzed by his students (e.g. Kenneth De Jong) in the USA, and at the same time, evolution strategies were invented in Germany by Ingo Rechenberg. The main differences between these two kinds of EAs are the method of creating the new generation and the typical representation form for individuals: it is bit-string for GAs and real vector for ESs. The two kinds of EAs also differ in the way genetic operators are applied.

There is a special kind of genetic algorithms, namely *genetic programming* (GP), introduced by John R. Koza [22]. The main invention of GPs is that branching structures can be evolved. Most of the methods are the same as in GAs, but there are special genetic operators designed for branching structures: e.g. recombination replaces subtrees of the selected individuals.

In the following, the characteristics of genetic algorithms, genetic programming and evolution strategies are presented in brief. At the end of the section, the possibilities of the visualization are discussed.

2.1 Genetic Algorithms

Genetic algorithms are the most popular sort of evolutionary algorithms, where the individuals are usually represented by a series of bits. The genetic operators are implemented in accordance with this representation form. Genetic algorithms have proven to be successful at searching multidimensional spaces in order to solve, or solve approximately, a wide variety of problems [13, 25]. Here follows the description of the two most important genetic operators for GAs: *mutation* and *crossover*.

Mutation randomly changes each bit of an individual with a certain probability. The change can be done by either flipping a bit or replacing its value with a newly generated random value. In both cases it is important that it is considered for each bit independently whether to change it or not.

In the case of GAs, the *recombination* operator always takes two parents and creates two descendants, thus it is usually called *crossover*. The main kinds of crossover are *point-based crossover* and *parametrized uniform crossover*. For *point-based crossover*, the crossover points (whose number is given) are chosen at random. The case when there is only one crossover point is called *single-point crossover*. After choosing the crossover points, the parts of the individuals between these points are exchanged. *Parametrized uniform crossover* exchanges each bit of the parent individuals with a given probability to create the descendants.

The process of *creating the new generation* for a GA is quite simple: first, a new empty population is created. Then, to ensure the monotonicity of the process, a number of best individuals in the previous generation is copied into the new pop-

ulation as determined by the elitism rate parameter. After that, the remaining places are filled out in the population by selecting two parent individuals from the old population, performing mutation and crossover on them, and inserting either one or both of the descendant individuals into the new population as necessary. These operations (from the selection to the insertion) are repeated in a loop until the new population has enough individuals.

The *selection method* is a very important part of genetic algorithms, since selection assures that the fitness values of the individuals are constantly increasing during the evolution process. Since there are a wide range of functions that can be optimized with genetic algorithms and these functions behave very differently, various selection methods have been developed to deal with them [27]. For example, if a function has many local optima and some of these optima are very close to the global optimum, then selection pressure should be kept low in order to explore the whole search space rather than founding one local optimum and get stuck at it. For easier functions, which are smooth and have no local optima, the selection pressure can be set high in order to achieve faster convergence. Selection pressure means a function of fitness value that determines the relationship between fitness values and the probability of an individual with that fitness value to get selected. The selection probability of an individual is usually proportional to its fitness value or rank in the population. Other constructs use only a subset of the population when selecting or apply more complicated transformation functions to the fitness values. Interactive selection is usually used when it is impossible to formalize an effective fitness function (e.g. in some design and shape recognition applications [2, 5, 14, 24, 34, 40]); here, the individuals are presented to the user and he/she can decide which of them are the most suitable solutions.

2.2 Genetic Programming

It is difficult and restrictive to represent hierarchies of dynamically varying size and shape with fixed length vectors. *Genetic programming* (GP) uses the same algorithms for creating the new generation and selecting individuals as genetic algorithms. The difference between GAs and GP is that GP uses a tree-like representation form for individuals, thus it provides a way to find a function or a computer program of unspecified size and shape to solve a problem [22].

Genetic programming has been successfully applied to problems such as classification [1] and pattern recognition [23, 33], generation of maximal entropy sequences of random numbers [21], Boolean function learning [11, 26], simultaneous architectural design [28] and training of neural networks [29].

GP's *genetic operators* work with sub-trees of the individuals. *Mutation* chooses a node of the tree and replaces the corresponding sub-tree with a new, randomly generated one, while *crossover* creates the offspring by exchanging randomly selected sub-trees of the parent individuals.

2.3 Evolution Strategies

Evolution strategies (ESs) are less popular than genetic algorithms, although they stand closer to the natural evolution since competition with their descendants is enabled for the parent individuals.

There are two kinds of evolution strategies, the so-called *comma and plus strategies*: $(\mu/\rho, \lambda)$ -ES and $(\mu/\rho + \lambda)$ -ES. Here μ , ρ and λ denote the population size, the number of parents used in recombination and the size of the selection pool, respectively. The *selection pool* is a temporary storage for individuals: offspring of the selected parents are put into it and the new generation is formed from the best μ individuals of the selection pool. The difference between the comma and plus strategies is that the plus strategy puts the old population (the parents) into the selection pool after generating λ individuals. Obviously, $\mu \leq \lambda$ must hold if the comma strategy is applied. There are special cases for ES, e.g. when ρ is set to 0 or 1 (or omitted) then recombination doesn't take place, only mutation is applied. Other special cases are $(1 + 1)$ -ES (hill climbing) and $(1, 1)$ -ES (random search). For ESs, the common *representation form of individuals* is a fixed length real vector. The genetic operators are developed in accordance with this specific representation form.

The *mutation* operator of evolution strategies is very similar to that of genetic algorithms: it changes each element of the real vector (i.e. each gene) with a certain probability. The difference originates from that the genes are real numbers, so they can be either multiplied or increased by a random value (the distribution of the value added to the gene is usually normal). The extent of this random value is controlled by the mutation rate parameter.

ES recombination takes ρ individuals as parents and produces one descendant of them. (Recall that GA's crossover takes two parent individuals and creates two descendants.) ES recombination methods can be classified by two aspects: there exist *discrete/intermediate* and *local/global* recombination methods; their detailed description with examples can be found in Section 5.3 of [16].

The algorithm for *creating the new generation* for an ES is the following: First λ individuals are created in the empty selection pool. To create a new individual, ρ parent individuals are selected *randomly* from the old population. Then recombination is performed on these individuals to get a descendant. After mutating the descendant, it is put into the selection pool. In the case of the plus strategy, the individuals from the old population are also put into the selection pool. Note that the random selection does not assure the convergence of the process, it is assured by forming the new generation from the best μ individuals of the selection pool.

In nature, it can be observed that populations of the same species are sometimes evolving separately, and after some generations they meet. In the field of evolution strategies, this phenomenon is realized by means of the so-called *meta-ES* method ([16], Subsection 5.4.5). In meta-ES, several populations of the same type are evolved separately for some generations, and these populations are modified by genetic operators. I.e. the populations are regarded as individuals (vectors of individuals), thus genetic operators can be applied on them. Mutation can be carried

out by randomly replacing some individuals in the population, and recombination can work as crossover works in GAs. The similar approach in genetic algorithms is called *island model*.

2.4 Possibilities of the Visualization

Visualizing an evolutionary algorithm is useful for *controlling* its run and *understanding* its behavior. Controlling includes the configuration and the interactive execution of the evolution process. The behavior can be analyzed by observing the operation of the selection and the genetic operators, the quality of the solutions found, the individuals' genotypes and phenotypes etc.

To show the internals of the process, basically the following *three techniques* can be applied:

Plots are suitable for displaying a smaller amount of numerical data like the values of a feature as a function of one or two other parameters. Depending on the number of the function parameters, two-dimensional or three-dimensional plots can be created.

Color coding is an efficient method to display larger amounts of numerical data in a tabular and still easily readable form. Here a two-dimensional table is created, the rows and columns being indexed by the discrete values of the two parameters and the cells representing the respective value by a color. A color is assigned to both the lowest and highest values in the table and intermediary values are represented by tones between these two colors.

Drawings can be used to display graphical objects such as the phenotypes of the evolved individuals. This way the changes and differences on the genotype level can be easily recognized as corresponding changes in the individuals' behavior in their evaluating environment.

When talking about visualization possibilities, one has to distinguish between the so-called *course* and *status visualization* methods, that is, between the ones that provide information about the *progress* and the *current state* of the process.

In the case of evolutionary algorithms, course visualization includes plots of particular fitness values, consumed system resources and the diversity of the population (e.g. standard deviation of the fitness values). The plots are usually drawn against generation number or, in the case of a steady-state GA, the number of evaluated individuals, but the used CPU time is a very good base for benchmarks, too.

The most useful color-coding progress visualization methods are those which show the best individuals' genotypes and the fitness values of all individuals of each generation. Though the first method is applicable only for fixed-length numerical chromosomes, together with the fitness graphs, it helps identifying the roles and importance of the single genes or gene groups. The latter view of the population shows somewhat more information about the fitness distribution than the deviation graphs.

In most cases, displaying information about the current status of an evolution process means showing some characteristics of the complete current generation. This information can be, for example, the genotypes or phenotypes of all individuals or just the occurring lowest and highest gene values.

Showing the phenotypes of individuals can be very productive when one needs to understand the connection between the genotypes and the phenotypes. However, being a completely problem-dependent visualization technique, it requires more implementation work from the user than just providing a fitness function.

A very important aspect of graphical data portrayal is the correct determination of the amount of the displayed information: the views should be enough for the user to be able to find the sought relations. On the other hand, they say that one figure is worth a thousand words; but the user should not be overwhelmed by an undigestable pack of knowledge.

3 The e_params Data Structure

This section gives a short description of a data structure that was designed to hold parameters of arbitrary objects such as various processes, data elements etc. Its main features are that *relationships* can be defined between the parameters and *conditions* and *restrictions* for the parameter values.

The data structure is designed in a way that the input and output of the functions are stored in easily readable text files, thus they can be modified with a plain text editor or script files.

e_params is implemented in *ANSI C* language for portability and simplicity reasons. It uses some elements of the *GLib² library* which is distributed under the *Free Software LGPL* and is available on UNIX, Win32 and OS/2 platforms. The current version of e_params is 0.14.

An extension has been implemented which enables the setting of the parameter values on a graphical user interface (GUI). This extension is written in *ANSI C* as well and it uses the *GIMP ToolKit (GTK²)*, which is available on several platforms including Linux and Win32 systems. Of course the e_params data structure can be used without the graphical extension.

The first application of the e_params data structure is related to evolutionary algorithms. The ordinary data types, possible conditions, restrictions and relationships are defined in a way that suits this purpose.

The domain of evolutionary algorithms requests that a list of *main parameters* (the values of which have to be given in every case) should be defined, and some ordinary types can have *dependent parameters* which have to be set iff the value of another parameter satisfies a condition. Moreover, *conditions* can be defined for some data types and certain parameters can *restrict* the possible values of other parameters.

The possible *data types* of the parameters include strings, file names, integer and real numbers and boolean values. A type called *OptionList* has been introduced for parameters which can have their values from a predefined finite set (*Options*).

²<http://www.gtk.org>

Each of the predefined values can have *dependent parameters* (which must be set only if the parameter is set to this option) and the options can also *restrict* the possible values of other OptionList parameters. Special types can also be defined for more sophisticated functionality.

Conditions can be assigned to numerical parameters by setting lower and/or upper boundaries for them. The value of the numerical parameter is valid iff it meets all the conditions assigned.

Restrictions are a kind of relationship between an Option and a parameter of the type OptionList: when an Option is assigned to a parameter as its value, the possible values of other OptionList parameters can be limited. For example, if the representation of the individuals in an evolution strategy is BitString, it doesn't make sense to compute the average of the bits, so the intermediate recombination cannot be selected as the recombination type. A *function* is provided for the data structure that *checks* whether the value of a given parameter *satisfies* its conditions and restrictions or not.

All parameters of a given object can have default values which are defined along with the parameters.

The definition of a parameters data structure is stored in a plain text file with the suffix ".ep" the format of which is given formally by a grammar in *Extended Backus-Naur Form* (EBNF, [41]) and context-sensitive restrictions. The files that store parameter values for an e_params parameter structure usually have the suffix ".epv". In such a file, the parameter values are stored in lines of the form "*parameter_name = parameter_value*". Special forms may be defined for special-type parameters such as arrays. Lines beginning with a hash mark are regarded as comments.

A *graphical extension* was implemented in order to provide an easy-to-use interface for setting the parameter values. It is realized using the *GIMP ToolKit*² because it is available in different platforms (e.g. Linux and Win32). From version 0.12, GTK version 1.1.4 is required. An important feature of the extension is that it *checks* the parameter values whether they satisfy the defined conditions and restrictions. The *dependent parameters* can be set up easily as well.

The form of the parameter setting dialog box can be seen in Figure 1. Each row shown in the table corresponds to one parameter. The second column of the table shows the parameter's "display name" (which can be different from the name used for the internal representation), and the value itself can be set using the widget placed into the third column. The type of this widget is determined according to the type of the parameter (for example, the value of a parameter of type OptionList can be set with a combo box).

If dependent parameters can be set to a parameter, then the *Parameters* button is enabled in the last column. When it is pressed, a new window appears offering modifications to the dependent parameters.

The first column of each row contains a hash mark which indicates the *correctness* of the parameter in that row. When the hash mark is *yellow*, then the parameter value had been changed since the last check and the new value has not been checked yet. A *green* hash mark indicates a correct value of the parameter and a *red* one

The figure shows two stacked dialog boxes for parameter settings. The top dialog box is titled "Dependent parameters of 'SubPopulations'" and contains the following settings:

- # Individual representation: Bitstring (with a checkmark and a "Parameters" button)
- # Optimization type: Minimization (with a checkmark and a "Parameters" button)
- # Algorithm type: Genetic algorithm (with a checkmark and a "Parameters" button)
- # Fitness value of a subpopulation: value of the best individual (with a checkmark and a "Parameters" button)
- # Mutation method for subpopulations: create random individuals (with a checkmark and a "Parameters" button)
- # Separation interval: 10

The bottom dialog box contains the following settings:

- # Recombination (crossover) method: none (with a checkmark and a "Parameters" button)
- # Recombination probability: 70.0 (with a slider and a "Parameters" button)

Both dialog boxes have "Okay", "Cancel", and "Check" buttons at the bottom.

Figure 1: The parameter settings dialog boxes of the `e_params` data structure. The figure shows the setting of the dependent parameters

indicates that the value is incorrect. The *Check* button can be pressed to perform a test of the values of the parameters. A check is performed automatically when the window is first displayed and when the *Okay* button is pressed. Parameters with incorrect values cannot be saved.

When the value of a parameter of type `OptionList` is changed to an `Option` that has defined restrictions to other `OptionList` parameters, then the combo boxes of the displayed restricted parameters are updated so that they will contain those options which are enabled by their restrictors. By pressing the *Browse...* button right to a file or directory name input field a standard file/directory selection dialog box appears in which the user can select a file/directory easily.

4 The GEA System

Kókai, Ványi and Tóth have been involved in evolving fractal images since 1997. The first attempt was to reproduce and improve Koza's results with *Lindenmayer systems* (*L-systems*) [18, 22]. This project was written in *Java* and did not use any general genetic programming libraries. Then it was realized that *L-systems* are capable of describing plants and these plants can be evolved by interactive evolution (the *TEvol* program, [19, 37, 38]). At the same time an ophthalmologist came up with the idea of describing the blood vessels of the eye with *L-systems*. This idea led to the *GREDEA* system [20, 39]. These two projects required the evolution of the rewriting rules of the *L-systems* as well as their parameters. The most suitable algorithm for the evolution of the rewriting rules is genetic programming, while the one for the parameter vectors are evolution strategies.

Since the *ANSI C++* programming language was used to implement *TEvol* and *GREDEA* and a programming library which dealt with both GPs and ESs could not be found at that time (in 1998), the design and implementation of a suitable system had begun. This system was later named *GEA* (Generic Evolutionary Algorithms Programming Library).

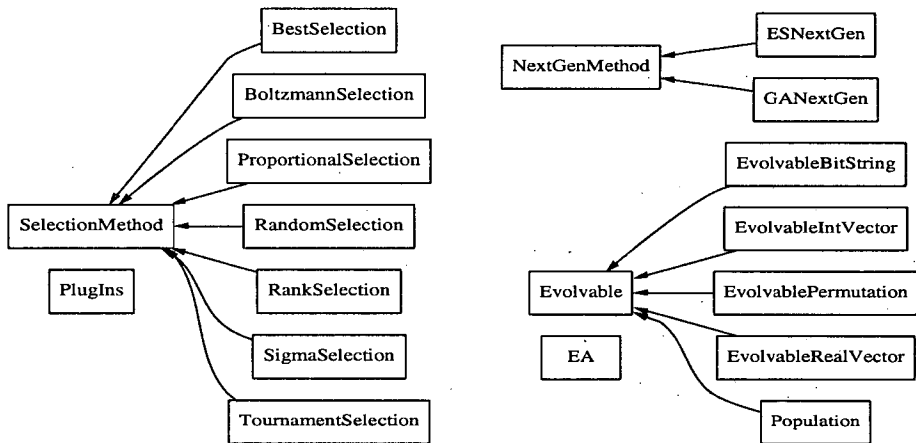


Figure 2: The class hierarchy of the *GEA* system

The class hierarchy of the current version of *GEA* can be seen in Figure 2. Already the first version contained the *Evolvable* abstract class which is the superclass of all evolvable objects, but at that time the integration of new selection methods and evolutionary algorithms was not easy to carry out. The latest version contains the abstract classes *SelectionMethod* and *NextGenMethod* as well, which define interfaces for selection methods and evolutionary algorithms. These enable the user of the system to easily expand it.

The latest version of *GEA* uses the so-called plug-in technology for the integration of

newly implemented classes. The subclasses of `SelectionMethod`, `NextGenMethod` and `Evolvable` have to be compiled and linked as shared libraries (*.so* files on Unix systems and *DLLs* under Windows). When the new plug-ins are registered in the parameter data structure of *GEA* (see Section 3), it will find and load them if necessary.

The application of the `e_params` data structure is also new in *GEA*. This data structure makes the extension of the system easier and provides a hierarchical structure of the parameters. Just as a sidenote, the system has currently 94 parameters (not all of which have to be set at the same time), which makes having a transparent interface to them reasonable.

Class `Evolvable` is the abstract superclass of all evolvable classes: it declares all the methods a class has to implement in order to become an evolvable class and implements a few basic functions. An `Evolvable` object represents one individual in the evolution process.

The *GEA* system uses three genetic operators which must be implemented in all evolvable classes: *Mutate*, *Crossover* and *Recombine*. Input/output and factory functions provide an interface for the transportation of the evolvable objects.

GEA has currently four *built-in representation forms*, namely for bit-strings, real vectors, integer vectors and permutations. The class that represents a *population* is also an evolvable class (that is, genetic operators can be applied on it), this makes experiments with meta-ES in *GEA* possible.

The abstract class `SelectionMethod` is the superclass of all implemented *selection methods* in *GEA*.

As it is explained in Section 2, evolutionary algorithms mostly differ in the way the individuals are represented and new generations are created. Various representation forms are available via the `Evolvable` abstract class and its subclasses. Different methods for creating a new generation are available in *GEA* through the `NextGenMethod` abstract class and its subclasses. Just like in the case of the selection methods, evolutionary algorithms are implemented as plug-ins and the required class is loaded at running time.

Currently, two *evolutionary algorithm frameworks* are available in the *GEA* system: `GANextGen` and `ESNextgen` implement genetic algorithms and evolution strategies as they are described in Subsections 2.1 and 2.3, respectively.

Class `EA` represents an *evolution process* in the *GEA* system. It has all methods at its disposal that are necessary to handle a population and create new generations. The constructor of the class receives an `e_params` data structure and according to the settings, it loads the necessary plug-ins and creates the initial population.

A common *plug-in handling interface* is provided to all classes which use shared libraries by class `PlugIns`. A static data member is used to keep account of the loaded shared libraries and a function can be used to look up a given symbol in a given shared library; the function loads the object file if needed.

The most important *problem-dependent function* in all evolution processes is the *fitness function*. In the *GEA* system, fitness functions are implemented as callbacks and are loaded from plug-ins, like every customizable part of the program code. The callback receives a pointer to an evolvable object as its parameter and should

return the result of the evaluation as a real number. Whether this value should be maximized or minimized is determined by the parameters of the evolution process. For some of the optimization problems, it is necessary to perform certain preparatory tasks before the start of the evolution process (e.g. the training and test data sets have to be loaded and preprocessed for a machine learning application). The data structures created by the preparator function and used for fitness calculation have to be properly destroyed after the optimization process has finished and sometimes the task requires maintaining operations between the generations. These tasks can be performed in *GEA* by so-called *preliminary*, *intermediary* and *posterior* functions.

After the problem-specific implementations (fitness function, in some cases special functions and/or individual representation) are ready, the optimization process can be started by typing

`GEA <parameter value file> <path to parameter structure> [shmid]`
into the command line. The command-line parameters are the following:

parameter value file Contains the values of the parameters of the evolution process.

path to parameter structure The name of the directory that contains the description of the parameter data structure of *GEA*.

shmid This optional argument is a so-called *shared memory identifier*. This is an integer number used to identify shared memory locations in the *Unix System V Interprocess Communication* system. When *GEA* is being run by *GraphGEA*, the calling graphical user interface allocates this shared memory and the two programs communicate through it. This feature is available only on Unix platforms.

When *GEA* is started, it performs the following tasks:

- loads the parameter structure file
- loads the parameter values
- processes the termination parameters of the EA
- processes the logging parameters of the EA, initializes logging facilities
- if an *shmid* is provided in the command line, initializes the communication with *GraphGEA*
- creates the evolution process
- runs the evolution process according to the termination parameters; during the run, manages logging and listens to the messages of the controlling graphical user interface
- after the evolution process had finished, properly frees the used resources and closes the log files

Since the input file of *GEA* is an easily readable and editable text file, the automation of performing several runs of the evolutionary algorithm with different parameters is very simple to carry out. Previous runs can be reconstructed by directly specifying the random seed of the process. Knowing the structure of the log files, the results of the run(s) can be extracted and converted to the desired format with standard text-processing tools. *GEA* is capable to dump the genotypes and the phenotypes of all individuals to given files in certain generation intervals or after the evolution process had finished. The genotype dumps can be used as *milestones* to start an evolution process later with a given initial population. For more information on the *GEA* system, see [35] and [36]. Usage examples can be found on the *GEA* home page.

5 GraphGEA

This section describes the *GraphGEA* program in detail. The system uses the graphical object set of the *GIMP ToolKit* (GTK²) which is written in the C programming language, thus this same language was used to implement *GraphGEA*. The functionality is presented beginning with the parameter settings, through the execution of the evolution process and closed with the visualization possibilities.

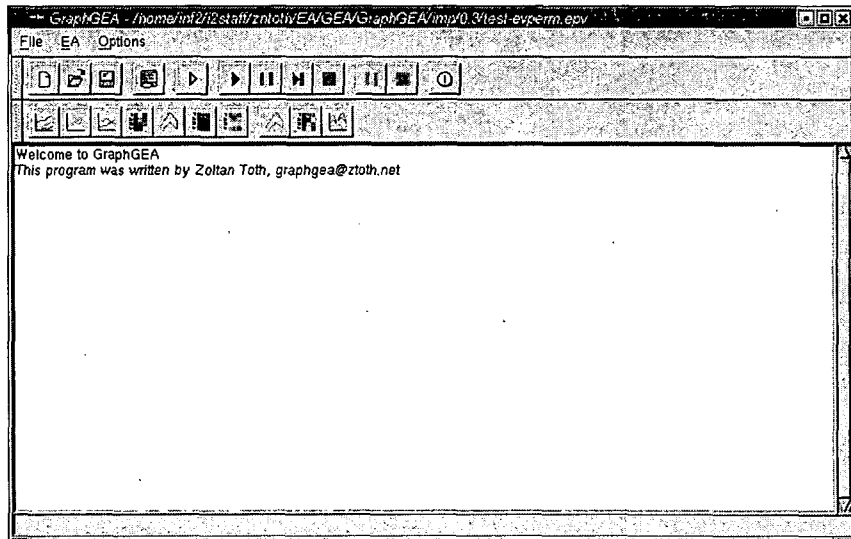



Figure 3: The main window of *GraphGEA*

The central window of the software is depicted in Figure 3. Below the menu bar, the main window of *GraphGEA* contains two rows of buttons (so-called toolbars), the upper row for managing parameter settings and controlling the evolution process, the lower one for showing and hiding the various visualization windows. The middle

of the window is occupied by a large text field where the messages of the application are written. Among others, these messages provide information about the actions between the graphical user interface and the underlying *GEA* process. Error reports are also printed here if one or more of the parameter values are invalid. A menu item of the Options menu serves for clearing the text field. A hint bar can be found at the bottom of the window. If the user moves the mouse over a button or a menu item, a short description of the associated function appears in this area.







5.1 Managing the Parameters

When the *GraphGEA* program starts, it loads the parameter structure definition and main parameter list of *GEA*. The first three buttons of the first toolbar realize the *New-Load-Save* functions known from many applications. The program keeps track of the changes of the parameter values and sends confirmation messages if non-saved information might be lost or used.

The  button brings up a dialog box of the *e_params* data structure (introduced in Section 3) with the main parameters of the evolution process. The main parameters are divided into three groups: the representation form of individuals, halting condition, applied genetic operators etc. belong to the first group. The second group contains the program-specific parameters such as the fitness function and plug-in file names, while the third group is for specifying logging options and log files.

5.2 Running the Evolution Process

After the parameters are set, the evolution process can be started and controlled with the buttons that resemble to those of CD or cassette players. Their functions are the following:

-  Starts the evolution process. The mechanism of the interaction between *GraphGEA* and *GEA* is described below.
-  If the evolution process is running, this button can be used to suspend it after the current generation has been processed. The suspended process can be resumed by pressing the button again.
-  If the evolution process is suspended, it can be executed generation by generation with this button, that is, this button proceeds one generation in the process. This enables the user to conveniently analyze the progress of the EA.
-  Causes the evolution process to stop after the current generation. After it is clicked, *GraphGEA* sends an appropriate signal to *GEA* and waits until it exits before enabling other actions for the user.
-  and  The two red buttons of the second group of controls can be used to pause/resume and stop the running evolution process immediately, i.e. without finishing the current generation and writing the results to the log files.

As it can be expected from every worthy application, the above listed buttons have their counterparts in the menu system of the program and they are enabled only if they are meaningful in actual state of the evolution process.

When the evolution is started, the graphical user interface invokes the *GEA* program as its child process and communicates with it during the run. The relationship and interaction between the two programs are depicted in Figure 4.

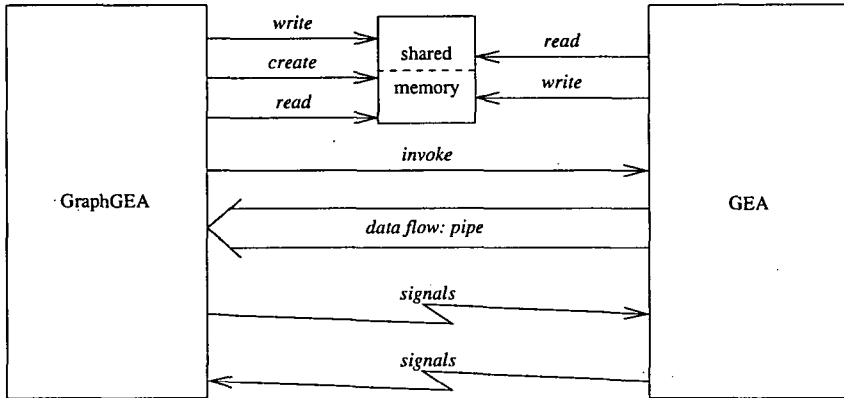



Figure 4: The interaction between *GraphGEA* and *GEA*

At the start of the evolution run, *GraphGEA* first checks whether the current parameter settings are correct and saved. If there are incorrect parameter values then it lists the error messages of the `e_params` data structure in its text area. In the case when the parameter settings have not been stored since the last changes, it asks the user if they should be written to disk before starting *GEA* or not.


Starting the *GEA* program takes the following steps: first, a shared memory area is requested from the operating system, the messages between the two programs are stored here before they are processed. Then *GraphGEA* creates a child process with the *fork* system call and the child invokes *GEA* with the necessary command-line arguments (see Section 4) using the *execvp* function. The parent process opens a pipe to the child and registers an event handler to manage its output (by default, *GEA* writes its log onto the standard output channel). Signal handlers are also registered by both programs, for they use the SIGUSR1 signal to let each other know about messages waiting on the shared memory area for processing.

Once the evolution process has started, the graphical user interface can send messages to it with system signals. Suspending and stopping *GEA* after the current generation and resuming a suspended run is done by placing the appropriate message identifier into the shared memory and sending a SIGUSR1 signal to the child process. *GEA* also sends a message to *GraphGEA* with the same mechanism each time a generation is ready. This is used for example at the step-by-step execution to enable/disable control buttons at the right time. Immediate suspend/resume and stop of the evolution process is done by sending SIGSTOP/SIGCONT and

SIGTERM signals, respectively. *GraphGEA* is watching the SIGCHLD signal, too, so that it knows when *GEA* exits.

The *off-line visualization* of an already finished evolution run can be initiated with the  button. For this, the parameter settings and the log file of the run are needed. When these two files are given, *GraphGEA* invokes a simple program (called *GEmul*) that echoes the log file to its standard output and communicates with the graphical user interface the same way as *GEA* does. In short, *GEmul* emulates the behavior of *GEA*, thus the suspension, step-by-step execution of the EA, etc. are all possible.

When a *complete reconstruction* of an evolution run is needed (a reason for this can be, for example, that the user wants to have more detailed logs), the original parameter settings are needed and the evolution process should be started with the random seed which was used in the original run (the used random seed is always printed into the log file).

After the work with *GraphGEA* is finished, the user can leave the program with the  button or by pressing *Ctrl-Q* on the keyboard.

5.3 The Visualization Plug-ins

The visualization options of the *GraphGEA* system are implemented as so-called plug-in modules (plug-ins for short). Plug-ins are compiled code segments, modules, which are not loaded by the operating system when the application is started, but the application itself can load them if it needs their functionalities. The most important advantages of plug-ins against traditional objects linked to the application are the following:

- Since they are stored in separate files (DLLs – dynamically loaded libraries – under Windows systems and .so – shared object – files in Unices), several applications can use the same files without the need of having the same compiled code stored several times on the hard disks.
- If an application does not need a certain module during a particular run, the code of that module doesn't have to be loaded and initialized, thus the start-up speed can be increased and the program can economize on system resources.
- Due to the standard interface of loading and using shared libraries, a part of a program can be improved by updating the plug-in file, thus avoiding the complete reinstallation.
- The standard interface also enables the easy and fast extension of applications, it is usually done by just copying the compiled code into a predefined directory and in some cases modifying configuration files.

Besides the advantages listed above, the generation of shared objects and their usage require only a very little implementation work from the program developer: in the compilation and linking, one only has to use a few additional command line

arguments of the linker, and loading and using the plug-ins in the main program make the call of only three simple library functions necessary.

The main reason of using plug-ins in the *GraphGEA* program is extendibility: new visualization plug-ins can be added with minimal modification of the existing program code. Each plug-in has a corresponding button in the second toolbar which shows and hides its visualization window. These buttons are enabled according to the successfulness of the loading and initialization of the plug-ins at start-up. *GraphGEA* looks up four functions (*create*, *init*, *new-data*, and *done*) in each loaded plug-in for the communication.

The evolution process (that is, the *GEA* program) runs as a child process of the graphical user interface and *GraphGEA* is reading its output from a pipe. Each time when the input handler function of the GUI gets a line from the pipe, it invokes the appropriate standard input handler function of each loaded plug-in. Each visualization method can decide whether the received information is relevant for its purposes or not and carry out the necessary actions (updating its database, executing certain drawing commands, etc.); for this reason it is very important to set the logging parameters of the evolution process correctly. If *GEA* does not print an information into the log (and to its standard output) then obviously this information will not be passed on to the plug-ins which might need them. On the other hand, if the user finds the output of one or more plug-ins irrelevant to his/her work, then turning off the corresponding logging options can be reasonable because it can increase the performance of the GUI. If the information turns out to be important in a later phase of the research, the evolution run can be reconstructed given the evolution parameters and the random seed are still available. The visualization windows with short descriptions are listed in Table 1.

5.3.1 Methods of Visualization

As it is described in Subsection 2.4, there are basically three different visualization methods discussed in this paper: plots, color coding, and drawings. Each of these three methods use the common plug-in interface but of course their behavior and look are different, so the implementation of some functions differs, too. Next, the look of the three plug-in window types and their functionality are discussed.

A *plot window* of *GraphGEA* is depicted in Figure 5. It is capable of showing several diagrams in one coordinate system, each of which can be shown and hidden individually with the checkboxes of the second toolbar. In the example, the best, mean and worst fitness values are depicted with different colors and the legend is displayed in the top-left corner of the plot. By default, the lower and upper boundaries of the X and Y axes are computed automatically according to the ranges of the shown values. This computation considers only the visible diagrams. The boundaries can be set manually in the first toolbar by unchecking the appropriate checkboxes and entering the values into the input lines next to them. The actual view of the diagrams can be saved in gnuplot format with the '*Save gnuplot*' button. The *gnuplot* program can convert its input into various well-know graphical formats, e.g. the encapsulated postscript file created from the plot shown in Figure 5 is

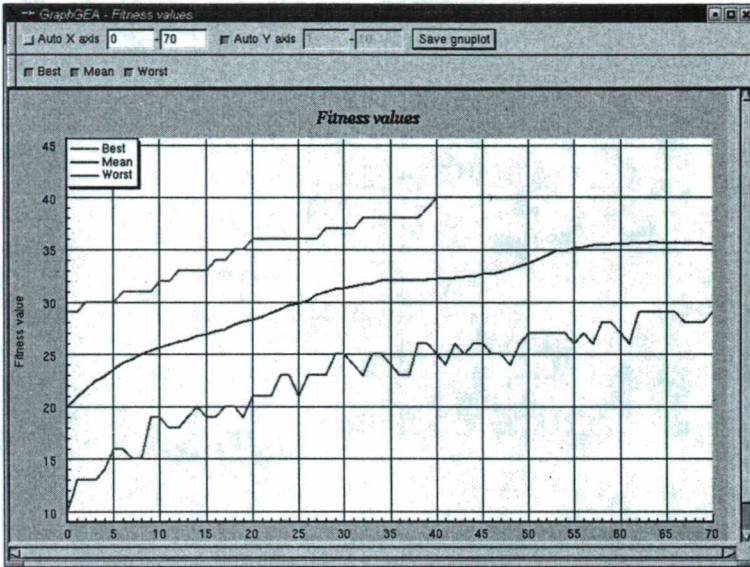
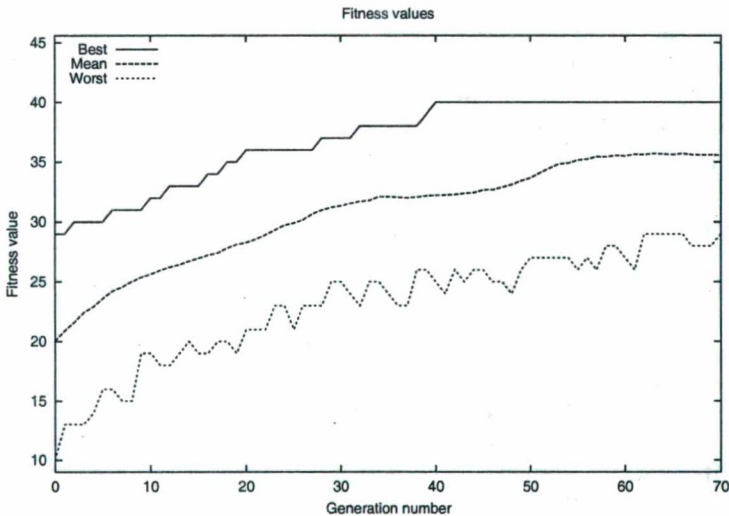
Figure 5: A plot window of *GraphGEA*

Figure 6: The gnuplot output of the plot window

displayed in Figure 6.

With the *color coding* technique, it is possible to depict a large amount of values in a transparent way: they are displayed with different colors, not with numbers. A color coding visualization window can be seen on Figure 7. Arbitrary numerical values can be shown in the form of a two-dimensional array with additional explanatory

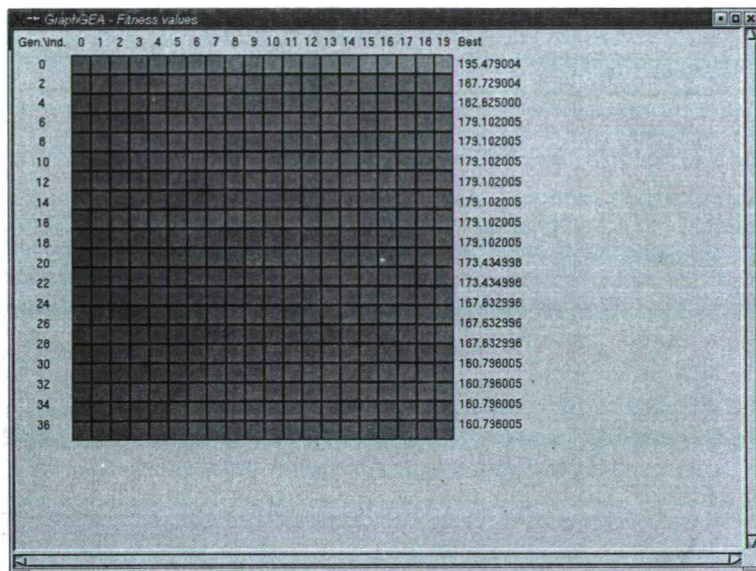


Figure 7: A color coding window of *GraphGEA*

columns on the left and the right hand side of the color matrix. In the current implementation, the lowest and highest displayed values can be specified directly or the plug-ins can compute them automatically. The specification or the automatic computation can be done either separately for each column or all columns can share the same limits.

There are two possibilities of displaying the individuals' *phenotypes* in the *GraphGEA* system: by printing the phenotypes as a series of strings into a text field or by using the drawing commands of the program. The phenotype visualization plug-ins choose between these two methods according to the representation form of the individuals. A window with a solution of the TSP problem can be seen on Figure 8. One individual is displayed at a time and the user can use the scrollbar at the top of the window to select from the available individuals. The initializer function creates and displays the appropriate drawing object by looking at the representation form of the individuals: if the representation is known as a drawable one (that is, its phenotype is printed as a series of drawing commands) then a drawing area is created, otherwise a text field will appear.

The set of *drawing commands* of *GraphGEA* is the following:

B x1 y1 x2 y2 This command determines the boundaries of the drawing area. The individual is drawn in a way that the graphical primitives within the boundaries are always visible in the plug-in window.

P x y Puts a point with coordinates (x, y) .

L x1 y1 x2 y2 Draws a line from $(x1, y1)$ to $(x2, y2)$.

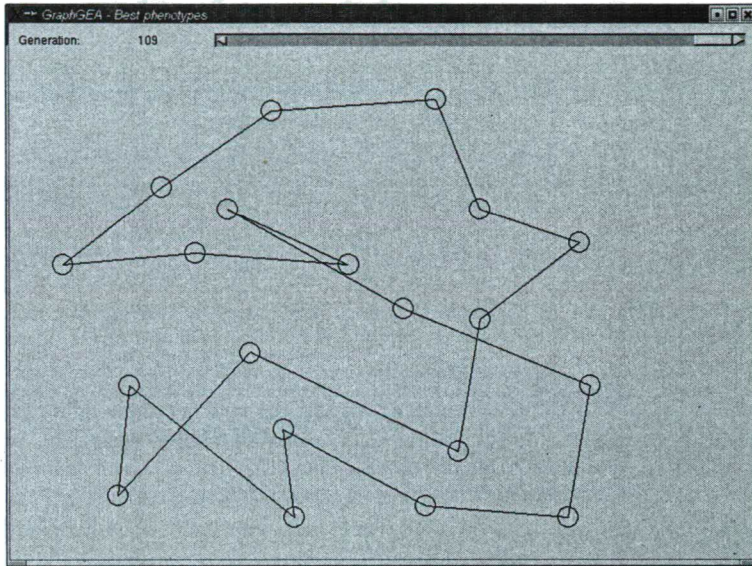


Figure 8: The phenotype of an individual drawn by *GraphGEA*

R x y w h f Draws a rectangle with the upper-left corner being in (x, y) , width w and height h . If f is equal to T then the rectangle will be filled.

A x y w h a1 a2 f Draws an arc. The upper-left corner will be at (x, y) , the width and height of the arc will be w and h , respectively. The starting angle of the arc is determined by $a1$, the length by $a2$ (that is, $a2$ is the ending angle relative to $a1$). The values of the angles should be between 0 and 360, 0 being at 12 O'clock, the positive direction is counter-clockwise. The last argument (f) determines the filling: $T = yes$, $F = no$.

Y n f x1 y1 x2 y2 ... xn yn Draws a polygon. First the number of vertices (n) is given, then the filling parameter, at the end follow the coordinates of the vertices.

S x y s Puts the string s at the coordinates (x, y) ; x and y are the left edge and the baseline of the string, respectively.

5.3.2 The Implemented Plug-ins

Table 1 shows the list of the currently available visualization plug-ins of *GraphGEA*. Besides the name, icon of the show/hide button and type of the visualization tools, a short description is also given.











Name	Type	Description
Fitness values 	plot, course	Diagrams of the best, mean and worst fitness values plotted against the generation number.
CPU times 	plot, course	A diagram showing the used CPU time of the evolution process plotted against the generation number.
Fitness variance 	plot, course	A diagram showing the variance of the fitness values in the population against the generation number.
Best genotypes 	color coding, course	A table containing the color coded gene values of the best individuals of the generations. The lowest gene values correspond to black cells, the highest gene values to white cells. The first and the last columns show the generation number and the fitness value of the depicted individual, respectively.
Best phenotypes 	draw, course	The phenotypes of the best individuals of the generations. The individual can be selected by the generation number.
Gene variances 	color coding, course	Shows the variances the of values of each gene in the population. Blue corresponds to low variance, red corresponds to high variance. The first and the last column contain the generation number and the fitness value of the best individual in the generation, respectively.
All fitness values 	color coding, course	The fitness values of all individuals are shown in one table. High fitness values with green, low values with red. The first and last columns of the matrix show the generation number and the fitness value of the best individual, respectively.
All phenotypes 	draw, status	Offers all phenotypes of the current generation for viewing. The individual can be selected by its position in the population.
All genotypes 	color coding, status	Displays all genotypes of the current generation. The low and high gene values are represented by white and brown colors, respectively. The first and last columns show the number of the individual and its fitness value.
Current gene values 	plot, course	The lowest, average and highest values of each gene are plotted against the gene number.

Table 1: The currently available visualization plug-ins of the *GraphGEA* system

6 Related Work

In this section some other EA visualizing/controlling tools and the differences between them and *GraphGEA* are discussed. It must be emphasized that the primary purpose of *GEA* is solving real world optimization problems and *GraphGEA* is a graphical user interface that supports analysis of the evolution process's behavior and education. *GraphGEA* does not affect the efficiency of the underlying evolution process.

The *EA Visualizer* [4] is a platform independent tool for running and visualizing evolutionary algorithms written in the Java programming language. It has a wide variety of convergence graphs and a special tool called *GraphDrawer* is provided to create various plots. Some of its disadvantages are that chromosomes can be depicted only in the case of binary representations and the phenotypes of the individuals can be drawn for some determined problems only, e.g. the traveling salesman problem (Figure 9). Since all individual representation forms in *GEA* have functions to output the genotypes of the individuals, these can be shown in every case. The internal drawing language of *GraphGEA* and the phenotype output of *GEA* enable depicting the phenotypes of solutions of any problem (see Figure 8). On the other hand, the *EA Visualizer* is able to handle multiple runs with different parameter settings. The evolutionary algorithms are implemented in Java and assembled from modules; this makes the system easily extendable, although genetic programming is not supported.

EvolVision [12] is a client-server based tool to visualize the output of *Mathematica notebooks* which use the *Evolvica* system [17]. The client-server architecture is very useful to make the EA process independent from the visualization tool, but *EvolVision* cannot control the run of the evolution process. It is able to perform off-line and on-line visualization as well and can depict any genomes and a range of various graphs. A plug-in interface is used for possible extensions. A disadvantage of the system is that it only realizes the results and has no real connection with the running evolution process. The graphical components of the Java language (*Swing*) are slow and require much memory and time for visualizing larger data sets.

GIGA [7] is what its name stands for: a Graphical user Interface for Genetic Algorithms. That is, only GAs are implemented and the evolution process can be controlled via the GUI to some extent; some parameters of the GA can be set in the control windows. It is able to do off-line and on-line visualization of some graphs and the algorithm's internals, but the latter figures are hard to read because the user has to find the crossover points and mutated genes himself, as these are not shown directly (see Figure 10). The phenotypic representation of the individuals is also available, but being completely problem dependent, this visualization has to be implemented by the user. The system is written in the C programming language using the Unix/X11 environment and the OSF/Motif GUI library, thus its portability is strongly bounded. It is possible to implement new algorithms for *GIGA*, but these must meet the quite strict restrictions of a given prototyping interface.

GeatBx [30] is another very promising visualization tool with various plots and

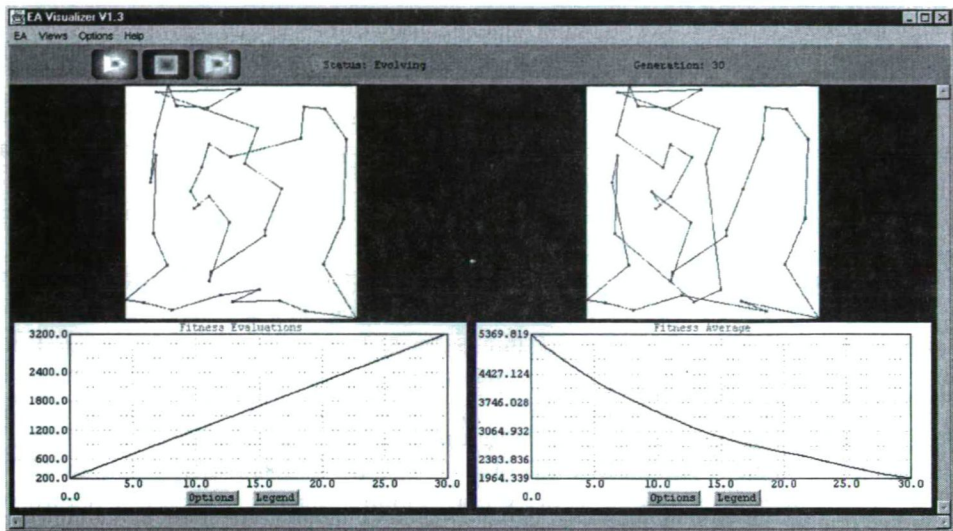


Figure 9: The *EA Visualizer* working on a TSP

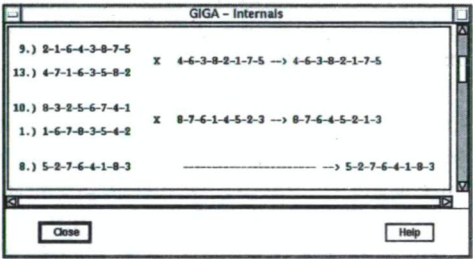


Figure 10: The 'internals' window of *GIGA*

graphs for depicting the *course* and the *state* of the running EA, but its disadvantage is that it is written in the *Matlab* computer algebra system, thus the user must know *Matlab* to use *GeatBx* efficiently. Besides, *Matlab* is a commercial software. The tool is able to do off-line and on-line visualization as well. GAs and ESs are implemented in the system but it is not able to visualize GPs and because of the lack of extendibility, the option to make experiments with these latter algorithms is completely missing. Only the genotypic representation of the individuals can be depicted, the phenotypes cannot be visualized with this tool.

Gonzo [6] is a tool for visualizing genetic algorithms written in LISP. The number of users of this system is strongly bounded because of the choice of the programming language, since LISP is not so widespread as C/C++ or Java. *Gonzo* is designed to explain the search behavior of the algorithm, so the search space and its representation stand in the center of this program. It can depict some graphs and plot

Name of the system	Supported algorithms	Language of implementation	Type of visualizations	Extension possible?	Interactive?	Off-line visualization?
GraphGEA	EAs	C/C++	Genotype, phenotype	yes	yes	yes
EA Visualizer	no GP	Java	Genotype, phenotype with restrictions	yes	yes	no
EvolVision	EAs	Mathematica, Java	Genotype, phenotype	yes	no	yes
GIGA	GAs	C, OSF/Motif	Genotype, phenotype	yes	partly	yes
GeatBx	no GP	MatLab	Genotype	no	yes	yes
Gonzo	GAs	LISP	Genotype	yes	yes	yes

Table 2: Comparison of the various visualization tools

how the gene values develop during the EA process (note that this technique is not applicable for genetic programming). Besides *GraphGEA*, this is the only system with total control over the running evolutionary algorithm: the user can start, stop, pause, resume the GA or execute it by generation steps.

The advantages and disadvantages of the described systems are summarized in Table 2.

7 Summary

In this document the *GraphGEA* system, a visualization extension of the *Generic Evolutionary Algorithms* programming library is presented. The first section covers the theoretical fundamentals of evolutionary algorithms. An evolution process has many, sometimes intricately interrelating parameters. A data structure for handling and extending this parameter structure is presented in Section 3. The *GEA* system is described in Section 4, while Section 5 deals with the *GraphGEA* system itself. Finally, a view on related work is given in Section 6.

GraphGEA has two main objectives: first, it helps the researchers to analyze and understand the search behavior of evolutionary algorithms, and second, it is a very good tool for students to get acquainted with these optimization methods. Since *GEA*, the underlying EA implementation, is an efficient and easy-to-use optimization utility, the graphical user interface can be used just to set all the parameters

of an optimization correctly, thus the GUI can be useful in solving real industrial optimization problems.

The graphical user interface can be divided into three main parts. Solving an optimization problem with an evolutionary algorithm always begins with the selection of the representation form of the individuals, the most suitable evolutionary algorithm and other parameters. *GraphGEA* offers very handy dialog boxes for setting all the parameters and it also assures that the values are correct. If one wants to analyze the optimization process, looking at the log files after the run is not always the best and most convenient way. The implemented software offers the possibility of the interactive execution of the evolution run, this way the user can suspend the process at any time and look at its course and status. The huge amount of numerical data describing an evolution run can be displayed by various visualization plug-ins in the *GraphGEA* system. The visualization windows provide a run-time look at the evolution process: the user can observe how the individuals change during the optimization, how much system resource is consumed, what is the diversity of the population, etc. Since the visualization methods are implemented as plug-ins and they have a common programming interface, it is very easy to expand the GUI with new methods.

Looking at the work done in the field of the visualization of evolutionary algorithms, the most important conclusion is that most of the available tools are very specific in terms of the implementation language and the range of suitable problems. Throughout the design of the *GEA* and *GraphGEA* systems, the two most important objectives were efficiency and applicability. This is the reason of the selection of the C and C++ programming languages and the application of the plug-in technology. Together with the used parameter structure, these make the programs able to solve and visualize a wide range of optimization problems.

References

- [1] M. Abramson and L. Hunter. Classification using cultural co-evolution and genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 249–254, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
- [2] K. Aoki, H. Takagi, and N. Fujimura. Interactive GA-based design support system for lighting design in computer graphics. In *International Conference on Soft Computing (IIZUKA '96)*, pages 533–536, Iizuka, Fukuoka, Japan, 1996. World Scientific.
- [3] W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors. *Proceedings of the Genetic and Evolutionary Computation Conference*, Orlando, Florida, USA, 13–17 July 1999. Morgan Kaufmann.

- [4] P. A. N. Bosman. EA visualizer.
<http://www.cs.ruu.nl/people/peterb/computer/ea/eavisualizer/EAVisualizer.htm>.
- [5] C. Caldwell and V. S. Johnston. Tracking a criminal suspect through face-space with a genetic algorithm. In *ICGA91*, pages 416–421, 1991.
- [6] T. D. Collins. *The Application of Software Visualization Technology to Evolutionary Computation: A Case Study in Genetic Algorithms*. Ph.D thesis, Knowledge Media Institute, The Open University, Milton Keynes, UK, 1998.
- [7] T. Dabs. Eine Entwicklungsumgebung zum Monitoring Genetischer Algorithmen. Master's thesis, University of Würzburg, 1994.
- [8] C. Darwin. *On the Origin of Species*. Murray, London, 1859.
- [9] L. Davis. Adapting operator probabilities in genetic algorithms. In *Proceedings of the Third ICGA*, pages 61–67. Morgan Kaufmann, 1989.
- [10] K. A. De Jong. An analysis of the behavior of a class of genetic adaptive systems. Ph.D thesis, University of Michigan, 1975.
- [11] S. Droste. Efficient genetic programming for finding good generalizing boolean functions. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 82–87, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.
- [12] T. Fühner and C. Jacob. Evolvision - an evolvisca visualization tool. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, Hans Michael Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, page 176, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.
- [13] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Reading, MA, 1989.
- [14] J. Graph and W. Banzhaf. Interactive evolution of images. In *International Conference on Evolutionary Programming*, 1995.
- [15] J. H. Holland. *Adaption of Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.
- [16] C. Jacob. *Principia Evolvica - Simulierte Evolution mit Mathematica*. Dpunkt Verlag, 1997.
- [17] C. Jacob. *Principia Evolvica - Simulierte Evolution mit Mathematica*, page 443. In [16], 1997.

- [18] G. Kókai, Z. Tóth, and R. Ványi. Application of genetic algorithms with more populations for Lindenmayer systems. In *Proceedings of the International Symposium on Engineering of Intelligent Systems, EIS'98*, pages 324–331. ICSC Academic Press, 1998.
- [19] G. Kókai, Z. Tóth, and R. Ványi. Evolving artificial trees described by parametric L-systems. In *Proceedings of the First Canadian Workshop on Soft Computing*, pages 1722–1728, Edmonton, Alberta, Canada, 9 # may 1999.
- [20] G. Kókai, R. Ványi, and Z. Tóth. Parametric L-system description of the retina with combined evolutionary operators. In Banzhaf et al. [3], pages 1588–1595.
- [21] J. R. Koza. Evolving a computer program to generate random numbers using the genetic programming paradigm. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 37–44, University of California - San Diego, La Jolla, CA, USA, 13-16 July 1991. Morgan Kaufmann.
- [22] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts, 1992.
- [23] J. R. Koza. Automated discovery of detectors and iteration-performing calculations to recognize patterns in protein sequences using genetic programming. In *Proceedings of the Conference on Computer Vision and Pattern Recognition*, pages 684–689. IEEE Computer Society Press, 1994.
- [24] D. Levine, M. Facelllo, and P. Hallstrom. Stalk: An interactive system for virtual molecular docking. *IEEE Science and Engineering*, 2/97:55–67, 1997.
- [25] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Artificial Intelligence. Springer-Verlag, 1992.
- [26] J. F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In Banzhaf et al. [3], pages 1135–1142.
- [27] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge Massachusetts, 1996.
- [28] U.-M. O'Reilly and G. Ramachandran. A preliminary investigation of evolution as a form design strategy. In C. Adami, R. K. Belew, H. Kitano, and C. E. Taylor, editors, *Proceedings of the Sixth International Conference on Artificial Life*, University of California, Los Angeles, 26-29 June 1998. MIT Press, Cambridge.
- [29] V. P. Plagianakos and M. N. Vrahatis. Training neural networks with 3-bit integer weights. In Banzhaf et al. [3], pages 910–915.
- [30] H. Pohlheim. Visualization of evolutionary algorithms – set of standard techniques and multidimensional visualization. In Banzhaf et al. [3], pages 533–540.

- [31] I. Rechenberg. *Evolutionsstrategien: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Fromman-Holzboog, Stuttgart, 1973.
- [32] H.-P. Schwefel. Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie. *Interdisciplinary Systems research (26)*, Birkh.user, Basel, 1977.
- [33] J. Sherrah. *Automatic Feature Extraction for Pattern Recognition*. PhD thesis, University of Adelaide, South Australia, July 1998.
- [34] J. R. Smith. Designing biomorphs with an interactive genetic algorithm. In *ICGA91*, pages 535–538, 1991.
- [35] Z. Tóth. *The Generic Evolutionary Algorithms Programming Library*. Master's thesis, University of Szeged, Szeged, Hungary, 2000.
- [36] Z. Tóth and G. Kókai. An evolutionary optimum searching tool. In *The Proceedings of the Fourteenth International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems (IEA/AIE-2001)*, volume 2070 of *LNAI*, pages 19–24, Budapest, Hungary, June 4–7 2001. Springer-Verlag.
- [37] Z. Tóth, G. Kókai, and R. Ványi. Interactive visual tree evolution. In *EIS2000 Second International ICSC Symposium on Engineering of Intelligent Systems, June 27 - 30, 2000 at the University of Paisley, Scotland, U.K.*, pages 384–390, 2000.
- [38] R. Ványi. *Modelling the Evolution of the Flora*. Bachelor's thesis (in Hungarian), József Attila University, Szeged, Hungary, 1998.
- [39] R. Ványi, G. Kókai, Z. Tóth, and T. Pető. Grammatical retina description with enhanced methods. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 193–208, Edinburgh, Apr. 15–16 2000. Springer-Verlag.
- [40] G. Venturini, M. Slimane, F. Morin, and J. P. A. de Beauville. On using interactive genetic algorithms for knowledge discovery in databases. In *ICGA97*, pages 696–703, 1997.
- [41] N. Wirth. What can we do about the unnecessary diversity of notation for syntatic definitions. *Communications of the ACM*, 20(11):822–823, Nov. 1977.

CONTENTS

Preface	207
<i>Tim Fühner and Gabriella Kókai</i> : Incorporating Linkage Learning into the GeLog Framework	209
<i>Gábor Gosztolya, András Kocsor, László Tóth, and László Felföldi</i> : Various Robust Search Methods in a Hungarian Speech Recognition System . . .	229
<i>Dávid Hanák</i> : Implementing Global Constraints as Graphs of Elementary Constraints	241
<i>István Katsányi</i> : On Implementing Relational Databases on DNA Strands . .	259
<i>Kornél Kovács and András Kocsor</i> : Various Hyperplane Classifiers Using Kernel Feature Spaces	271
<i>Balázs Polgár and Endre Selényi</i> : Probabilistic Diagnostics with P-Graphs .	279
<i>Raluca Oana Scarlatescu</i> : Programming by steps	293
<i>Paula Steinby</i> : Two Content Protection Schemes for Digital Items	315
<i>Attila Tanács and Attila Kuba</i> : Evaluation of a Fully Automatic Medical Image Registration Algorithm Based on Mutual Information	327
<i>Zoltán Tóth</i> : A Graphical User Interface for Evolutionary Algorithms	337